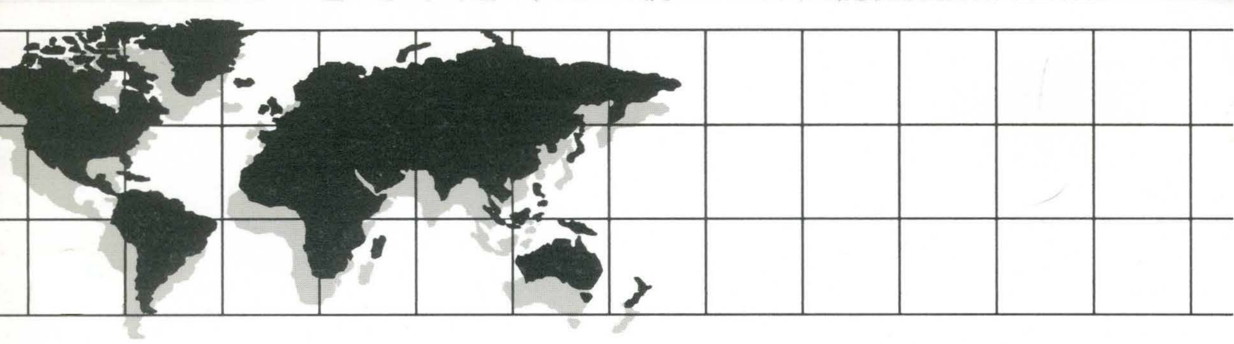

CTOS Procedural Interface

Reference Manual

Volume 3

Operations S through Z

UNISYS

UNISYS

CTOS® Procedural Interface

Reference Manual

Volume 3 Operations S through Z

This volume has been updated for CTOS III with pages from Update 1, 4357 4714-110

CTOS III 1.0	August 1992
CTOS II 3.4	
CTOS I 3.4	
CTOS/XE 3.4	
Development Utilities 12.2	
Standard Software 12.2	
Video Access Method 4.0	Printed in USA
Priced Item	4357 4565-100

Contents

List of Figures

List of Tables

SbPrint (psbString)

NOTE: This operation does not return a status code.

Description

SbPrint prints a string in which the first byte is the size of the string. The string is printed to the video or other device as specified by the NPrint and PutChar operations.

Procedural Interface

SbPrint (psbString)

where

psbString

points to a character string in which the first byte is the size of the string.

Request Block

SbPrint is an object module procedure.

This page intentionally left blank

ScanToGoodRsRecord (pRswa, qbSkipMax, pLfaScanStartRet, pqbRet): ercType

Description

ScanToGoodRsRecord should be called after ReadRsRecord returns status code 3607 ("Malformed record") or after a disk error occurs while reading the open RSAM file identified by the memory address of the Record Sequential Work Area. ScanToGoodRsRecord searches the sectors of the RSAM file until a valid record header is found. The double-check byte of the record found and the header of the following record are then checked, and if they are valid, the RSAM file is positioned to the record found. If the RSAM file is also a Direct Access Method (DAM) file, that is, a file of fixed-length records, record headers are only searched for at the positions where they can occur. These positions are computed by simple arithmetic involving the record length.

ScanToGoodRsRecord reads every sector in the area scanned, so that sectors of the file that were damaged are detected and skipped.

Procedural Interface

ScanToGoodRsRecord (pRswa, qbSkipMax, pLfaScanStartRet, pqbRet): ercType

where

pRswa

is the memory address of the same Record Sequential Work Area that was supplied to OpenRsFile.

qbSkipMax

is a 32-bit unsigned integer (the maximum number of bytes to skip while scanning).

pLfaScanStartRet

is the memory address of a logical file address where the byte offset in the RSAM file of the first byte skipped is returned.

pqbRet

is the memory address of a 32-bit unsigned integer where the number of bytes skipped is returned.

Request Block

ScanToGoodRsRecord is an object module procedure.

ScreenPrintVersion (pbVerInfo, cbVerInfo, pcbRet): ercType

Description

ScreenPrintVersion returns a structure that identifies the version of the installed Screen Print Service.

Procedural Interface

ScreenPrintVersion (pbVerInfo, cbVerInfo, pcbRet): ercType

where

pbVerInfo
cbVerInfo

describe a memory area to which the structure containing the version is written. The format of the structure is as follows:

Offset	**Field**	**Size (Bytes)**
0	version	2

pcbRet

is the memory address of a word containing the count of bytes returned.

Request Block

scbRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8229
12	reserved	6	
18	pbVerInfo	4	
22	cbVerInfo	2	
24	pcbRet	4	
28	scbRet	2	2

ScrollFrame (iFrame, iLineStart, iLineMax, cLines, fUp): ercType

NOTE: When using standard VGA on EISA/ISA-bus workstations, this operation may display some attributes (specifically, bold, underlining, struckthrough, and blinking) differently than expected. System configuration options allow you to substitute color for these attributes. See the CTOS System Administration Guide *for more information.*

Description

ScrollFrame scrolls the specified portion of the specified frame up or down by the specified number of lines. Vacated lines are replaced by blank lines. The portion to scroll begins at *iLineStart* and extends down to, but does not include, *iLineMax*. It is scrolled up/down by cLines and the bottommost/topmost *cLines* lines of the scrolled area are filled with nulls (character code 0). *fUp* specifies the direction of the scroll. A value of TRUE for *iLineStart* or *iLineMax* specifies an imaginary line just below the bottom of the frame.

For example, to scroll an entire frame up by one line, specify:

iLineStart = 0
iLineMax = TRUE
cLines = 1
fUp = TRUE

To open a two-line space at line 4 (that is, lines 4 and 5 become blank) by scrolling the frame down, specify:

iLineStart = 4
iLineMax = TRUE
cLines = 2
fUp = FALSE

To close the two-line space again, by scrolling the frame up (leaving the bottom two lines blank), specify:

iLineStart = 4
iLineMax = TRUE
cLines = 2
fUp = TRUE

Procedural Interface

ScrollFrame (iFrame, iLineStart, iLineMax, cLines, fUp): ercType

where

iFrame

specifies the frame (word). A value of 255 is treated as a special case and is reserved for internal use only.

iLineStart

is the line (word) at the top of the area to scroll.

iLineMax

is the line (word) just below the area to scroll. If *iLineMax* is greater than the line number at the end of the frame, no error is returned. Rather, *iLineMax* is treated as the end of the frame.

cLines

is the number (word) of lines by which to scroll. If *cLines* is greater than the number of lines in the frame, no error is returned. Rather, every line in the frame is replaced by a blank line.

fUp

is a flag (byte) that specifies the direction of the scroll. It is TRUE for scroll up or FALSE for scroll down.

Request Block

ScrollFrame is a system-common procedure.

This page intentionally left blank.

ScrubFile (fh): ercType

Caution: *This operation is for* **restricted use only.**

Description

ScrubFile completely erases file information from a volume. It writes 00h to every byte used by the file being deleted and sets the file length to 0 sectors. Files deleted using ScrubFile cannot be recovered using an Undelete operation.

Unlike ScrubFile, the **Delete** command (described in the *CTOS Executive Reference Manual*) only deletes the pointers to the file data. Because the file data still remains on the disk, it can be recovered.

Procedural Interface

ScrubFile (fh): ercType

where

fh

is the file handle returned from an OpenFile operation. The file must be open in modify mode.

Request Block

ScrubFile is an object module procedure.

ScsiCdbDataIn (pathHandle, wTimeout, pCdb, sCdb, pDataInRet, sDataInMax, pCbDataRet, pStatusRet, pExtSenseRet, sExtSenseMax): ercType

Description

ScsiCdbDataIn sends a SCSI command descriptor block to the SCSI target and LUN specified by the SCSI path handle and provides for an (optional) DATA IN phase for the transfer of information from the SCSI device.

ScsiCdbDataIn may not be used with paths opened in SCSI Manager target mode; status code 385 ("Invalid SCSI path handle") is returned in this case.

Procedural Interface

ScsiCdbDataIn (pathHandle, wTimeout, pCdb, sCdb, pDataInRet, sDataInMax, pCbDataRet, pStatusRet, pExtSenseRet, sExtSenseMax): ercType

where

pathHandle

is the SCSI path handle (word) returned by a ScsiOpenPath operation.

wTimeout

is a word that specifies the timeout value (in tenths of a second) for the operation. Zero indicates that the default timeout for the SCSI path should be used.

pCdb
sCdb

describe a 6-, 10-, or 12-byte SCSI command descriptor block.

pDataInRet
sDataInMax

describe the buffer area available to receive information sent from the SCSI device during the DATA IN phase.

pCbDataRet

is the address of a word to which the actual count of bytes of data transferred from the SCSI device during the DATA IN phase is returned.

pStatusRet

is the address of a byte to which status from the target is returned. (The status byte values are contained in the ANSI SCSI standard.)

pExtSenseRet
sExtSenseMax

describe an area where extended sense from the target is returned if the operation specified by ScsiCdbDataIn completes with a target status of CHECK CONDITION. If the SCSI Manager is unable to obtain the sense data, status code 395 ("Sense data unavailable") is returned. The sense data information is valid only if status code 0 ("ercOK") is returned by ScsiCdbDataIn and the target status is CHECK CONDITION. If *pExtSenseRet* is 0 or *sExtSenseMax* is 0, the SCSI Manager does not attempt to retrieve sense data. In this case, the sense data may still be available. Provided no intervening commands have been made by any client of the SCSI Manager to the SCSI device, a ScsiRequestSense operation may be used to obtain the data.

Request Block

sCbDataRet is always 2 and *sStatusRet* is always 1.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	1
3	nRespPbCb	1	4
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	360
12	pathHandle	2	
14	reserved	2	
16	wTimeout	2	
18	pCdb	4	
22	sCdb	2	
24	pDataInRet	4	
28	sDataInMax	2	
30	pCbDataRet	4	
34	sCbDataRet	2	2
36	pStatusRet	4	
40	sStatusRet	2	1
42	pExtSenseRet	4	
44	sExtSenseMax	2	

This page intentionally left blank

ScsiCdbDataInAsync (pathHandle, wTimeout, pCdb, sCdb, pDataInRet, sDataInMax, pExtSenseRet, sExtSenseMax, pRq, exchangeReply): ercType

Description

ScsiCdbDataInAsync initiates the transmission of a SCSI command descriptor block to the SCSI target and LUN specified by the SCSI path handle and provides for an (optional) DATA IN phase for the transfer of information from the SCSI device. ScsiWaitCdbAsync must be called to check the completion status of the operation.

ScsiCdbDataInAsync is a library procedure that provides an interface to the ScsiCdbDataIn request. (See ScsiCdbDataIn.) The information returned by the ScsiCdbDataIn arguments *pCbDataRet* and *pStatusRet* is not available when the SCSI operation is initiated but is returned when the ScsiWaitCdbAsync operation is performed.

ScsiCdbDataInAsync may not be used with paths opened in SCSI Manager target mode; status code 385 ("Invalid SCSI path handle") is returned in this case.

Procedural Interface

ScsiCdbDataInAsync (pathHandle, wTimeout, pCdb, sCdb, pDataInRet, sDataInMax, pExtSenseRet, sExtSenseMax, pRq, exchangeReply): ercType

where

pathHandle

is the SCSI path handle (word) returned by a ScsiOpenPath operation.

wTimeout

is a word that specifies the timeout value (in tenths of a second) for the operation. Zero indicates that the default timeout for the SCSI path should be used.

pCdb
sCdb

describe a 6-, 10-, or 12-byte SCSI command descriptor block.

pDataInRet
sDataInMax

describe the buffer area available to receive information sent from the SCSI device during the DATA IN phase.

pExtSenseRet
sExtSenseMax

describe an area where extended sense from the target is returned if the call to ScsiWaitCdbAsync indicates the operation specified by ScsiCdbDataInAsync completed with a target status of CHECK CONDITION. The sense data information is valid only if status code 0 ("ercOK") is subsequently returned by ScsiWaitCdbAsync and the target status is CHECK CONDITION. If the SCSI Manager is unable to obtain the sense data, status code 395 ("Sense data unavailable") is returned. If *pExtSenseRet* is 0 or *sExtSenseMax* is 0, the SCSI Manager does not attempt to retrieve sense data. In this case, the sense data may still be available. Provided no intervening commands have been made by any client of the SCSI Manager to the SCSI device, a ScsiRequestSense operation may be used to obtain the data.

pRq

is the address of a 64-byte area to be used as a work space by the ScsiCdbDataInAsync and ScsiWaitCdbAsync operations.

exchangeReply

is an exchange provided by the user for the exclusive use of ScsiCdbDataInAsync and ScsiWaitCdbAsync.

Request Block

ScsiCdbDataInAsync is an object module procedure that provides an interface to the ScsiCdbDataIn request (see ScsiCdbDataIn).

This page intentionally left blank

ScsiCdbDataOut (pathHandle, wTimeout, pCdb, sCdb, pDataOut, sDataOut, pCbDataRet, pStatusRet, pExtSenseRet, sExtSenseMax): ercType

Description

ScsiCdbDataOut sends a SCSI command descriptor block to the SCSI target and LUN specified by the SCSI path handle and provides for an (optional) DATA OUT phase for the transfer of information to the SCSI device.

ScsiCdbDataOut may not be used with paths opened in SCSI Manager target mode; status code 385 ("Invalid SCSI path handle") is returned in this case.

Procedural Interface

ScsiCdbDataOut (pathHandle, wTimeout, pCdb, sCdb, pDataOut, sDataOut, pCbDataRet, pStatusRet, pExtSenseRet, sExtSenseMax): ercType

where

pathHandle

is the SCSI path handle (word) returned by a ScsiOpenPath operation.

wTimeout

is a word that specifies the timeout value (in tenths of a second) for the operation. Zero indicates that the default timeout for the SCSI path should be used.

pCdb
sCdb

describe a 6-, 10-, or 12-byte SCSI command descriptor block.

pDataOut
sDataOut

describe the buffer area that contains the information to be sent to the SCSI device during the DATA OUT phase.

pCbDataRet

is the address of a word to which the actual byte count of data transferred to the SCSI device during the DATA OUT phase is returned.

pStatusRet

is the address of a byte to which status from the target is returned. (The status byte values are contained in the ANSI SCSI standard.)

pExtSenseRet
sExtSenseMax

describe an area where extended sense from the target is returned if the operation specified by ScsiCdbDataOut completes with a target status of CHECK CONDITION. If the SCSI Manager is unable to obtain the sense data, status code 395 ("Sense data unavailable") is returned. The sense data information is valid only if status code 0 ("ercOK") is returned by ScsiCdbDataOut and the target status is CHECK CONDITION. If *pExtSenseRet* is 0 or *sExtSenseMax* is 0, the SCSI Manager does not attempt to retrieve sense data. In this case, the sense data may still be available. Provided no intervening commands have been made by any client of the SCSI Manager to the SCSI device, a ScsiRequestSense operation may be used to obtain the data.

Request Block

sCbDataRet is always 2 and *sStatusRet* is always 1.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	2
3	nRespPbCb	1	3
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	361
12	pathHandle	2	
14	reserved	2	
16	wTimeout	2	
18	pCdb	4	
22	sCdb	2	
24	pDataOut	4	
28	sDataOut	2	
30	pCbDataRet	4	
34	sCbDataRet	2	2
36	pStatusRet	4	
40	sStatusRet	2	1
42	pExtSenseRet	4	
44	sExtSenseMax	2	

This page intentionally left blank

ScsiCdbDataOutAsync (pathHandle, wTimeout, pCdb, sCdb, pDataOut, sDataOut, pExtSenseRet, sExtSenseMax, pRq, exchangeReply): ercType

Description

ScsiCdbDataOutAsync initiates the transmission of a SCSI command descriptor block to the SCSI target and LUN specified by the SCSI path handle and provides for an (optional) DATA OUT phase for the transfer of information to the SCSI device. ScsiWaitCdbAsync must be called to check the completion status of the operation.

ScsiCdbDataOutAsync is a library procedure that provides an interface to theScsiCdbDataOut request. (See ScsiCdbDataOut.) The information returned by the ScsiCdbDataOut arguments *pCbDataRet* and *pStatusRet* is not available when the SCSI operation is initiated but is returned when the ScsiWaitCdbAsync operation is performed.

ScsiCdbDataOutAsync may not be used with paths opened in SCSI Manager target mode; status code 385 ("Invalid SCSI path handle") is returned in this case.

Procedural Interface

ScsiCdbDataOutAsync (pathHandle, wTimeout, pCdb, sCdb, pDataOut, sDataOut, pExtSenseRet, sExtSenseMax, pRq, exchangeReply): ercType

where

pathHandle

is the SCSI path handle (word) returned by a ScsiOpenPath operation.

wTimeout

is a word that specifies the timeout value (in tenths of a second) for the operation. Zero indicates that the default timeout for the SCSI path should be used.

pCdb
sCdb

describe a 6-, 10-, or 12-byte SCSI command descriptor block.

pDataOut
sDataOut

describe the buffer area that contains the information to be sent to the SCSI device during the DATA OUT phase.

pExtSenseRet
sExtSenseMax

describe an area where extended sense from the target is returned if the call to ScsiWaitCdbAsync indicates the operation specified by ScsiCdbDataOutAsync completed with a target status of CHECK CONDITION. The sense data information is valid only if status code 0 ("ercOK") is subsequently returned by ScsiWaitCdbAsync and the target status is CHECK CONDITION. If the SCSI Manager is unable to obtain the sense data, status code 395 ("Sense data unavailable") is returned. If *pExtSenseRet* is 0 or *sExtSenseMax* is 0, the SCSI Manager does not attempt to retrieve sense data. In this case, the sense data may still be available. Provided no intervening commands have been made by any client of the SCSI Manager to the SCSI device, a ScsiRequestSense operation may be used to obtain the data.

pRq

is the address of a 64-byte area to be used as a work space by the ScsiCdbDataOutAsync and ScsiWaitCdbAsync operations.

exchangeReply

is an exchange (word) provided by the user for the exclusive use of ScsiCdbDataOutAsync and ScsiWaitCdbAsync.

Request Block

ScsiCdbDataOutAsync is an object module procedure that provides an interface to the ScsiCdbDataOut request (see ScsiCdbDataOut).

This page intentionally left blank

ScsiClosePath (pathHandle): ercType

Description

ScsiClosePath closes the virtual path specified by the SCSI path handle. A SCSI path may not be closed unless all pending operations for the path have been completed.

Procedural Interface

ScsiClosePath (pathHandle): ercType

where

pathHandle

is the SCSI path handle (word) returned by a ScsiOpenPath operation.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	2	2
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	358
12	pathHandle	2	

This page intentionally left blank

ScsiManagerNameQuery (pbProcessorName, cbProcessorName, pSbScsiManagerNameRet, sSbScsiManagerNameMax): ercType

Description

ScsiManagerNameQuery returns the name of the SCSI Manager at the specified processor.

Procedural Interface

ScsiManagerNameQuery (pbProcessorName, cbProcessorName, pSbScsiManagerNameRet, sSbScsiManagerNameMax): ercType

where

pbProcessorName
cbProcessorName

describe the processor upon which the SCSI Manager is executing. If these parameters are 0, the name of the SCSI Manager at the local processor is queried. Otherwise, the request is routed to the processor specified.

pSbScsiManagerNameRet
sSbScsiManagerNameMax

describe a memory area for the return of the SCSI Manager name string. The size of the string is returned in the first byte; the remaining bytes contain the name. The maximum string length returned is 12 characters plus the length byte.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	2	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	363
12	reserved	6	
18	pbProcessorName	4	
22	cbProcessorName	2	
24	pSbScsiManagerNameRet	4	
28	sSbScsiManagerNameMax	2	

ScsiOpenPath (pbScsiManagerName, cbScsiManagerName, hostAdapter, targetId, LUN, pPathHandleRet, pPathParameters, sPathParameters, pbPassword, cbPassword, mode): ercType

Description

ScsiOpenPath establishes a virtual circuit connection, or path, between the caller and a SCSI device. The specified SCSI device does not have to exist nor does it have to be online for a path to be created. Subsequent SCSI operations return appropriate status codes if no SCSI device is present at the other end of the path. The path handle returned by ScsiOpenPath is used to refer to the SCSI device in subsequent operations such as ScsiCdbDataIn, ScsiCdbDataOut, or ScsiRequestSense.

ScsiOpenPath may also be used to enable the SCSI Manager target mode function for a specified LUN.

Procedural Interface

ScsiOpenPath (pbScsiManagerName, cbScsiManagerName, hostAdapter, targetId, LUN, pPathHandleRet, pPathParameters, sPathParameters, pbPassword, cbPassword, mode): ercType

where

pbScsiManagerName
cbScsiManagerName

describe the name of the SCSI Manger to which requests are to be routed. If the name of the SCSI Manager is not specified in the ScsiOpenPath call, no routing is performed.

hostAdapter

is the number (byte), 0 to *n*, of the SCSI host adapter. For example, a Series 286i workstation or a Series 386i or B39 workstation with an additional HSD–xxx or B25–xxx upgrade module has two SCSI host adapters: one is internal to the workstation, and the the other is contained in the upgrade module. They are referenced as 0 and 1, respectively.

targetId

is the target ID (byte), 0 through 7, of the SCSI target to which SCSI operations are to be directed. The target ID is usually selected by switches or jumpers on the device itself. Alternatively, *targetId* may be 0FFh, which indicates that the path handle returned is to be used for SCSI Manager target mode operations (such as ScsiTargetDataReceive, ScsiTargetDataTransmit or ScsiTargetCdbCheck).

LUN

is the logical unit number (byte) at the indicated SCSI target to which SCSI operations are to be directed. Many so–called embedded SCSI devices contain only one logical unit (LUN 0) although the LUN may be any value in the range of 0 through 7. For SCSI Manager target mode operations, the LUN specifies which logical unit of the local processor device is to be enabled.

pPathHandleRet

is the address of a word to which the SCSI path handle is returned. Any additional SCSI operations to be performed on the SCSI device uniquely identified by this virtual path must use this handle.

pPathParameters
sPathParameters

describe a path parameters block, as defined below:

Offset	Field	Size (bytes)	Description
0	fNoDisconnect	1	TRUE means the SCSI device must remain connected to the bus for the duration of a command being processed.
1	bSynchronousOffset	1	Is a read only field with the following values: 0 = asynchronous 1–255 = value of synchronous offset
2	wSynchronousPeriod	2	Is a read only field with the following values: 0 = asynchronous 1–1023 = value of synchronous period in nanoseconds
4	wDefaultTimeout	2	Is the default time limit (in tenths of a second) within which a SCSI operation must complete before error recovery procedures are initiated.
6	reserved	10	

Path parameters for a path opened for target mode operations (*targetId* is 0FFh) are read only. Specifying parameters in this case has no effect.

pbPassword
cbPassword

describe the password that authorizes opening the SCSI path to access the specified SCSI device. If no other paths (except peek mode paths) to the SCSI device exist, this password becomes the access password for the device and remains in effect until this path is closed.

mode

is a word that describes access, which can be exclusive, shared, or peek. *mode* is indicated by 16-bit values representing the ASCII constants mx (exclusive mode), ms (shared mode), or mp (peek mode). In these ASCII constants, the first character (m) is the high-order byte, and the second character is the low-order byte.

Access in exclusive mode guarantees that no other users are able to send SCSI commands to the device. If a path to a SCSI device exists in exclusive mode, any attempts to establish additional paths cause status code 219 ("Access denied") to be returned.

Access in shared mode allows more than one user to send SCSI commands to the device. If one or more paths to a SCSI device already exist in shared mode, any attempts to establish path(s) in exclusive mode cause status code 219 ("Access denied") to be returned.

Access in peek mode may be shared with other users who have established a path in shared mode. If, however, the user that originally established a path to a SCSI device in peek mode attempts to perform a SCSI operation after a path has been established to the same device in exclusive mode, the original peek mode path handle is no longer usable, and status code 385 ("Invalid SCSI path handle") is returned.

Request Block

sPathHandleRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	2	6
2	nReqPbCb	1	3
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	355
12	hostAdapter	1	
13	targetID	1	
14	LUN	1	
15	reserved	1	
16	mode	2	
18	pbScsiManagerName	4	
22	cbScsiManagerName	2	
24	pbPassword	4	
28	cbPassword	2	
30	pPathParameters	4	
34	sPathParameters	2	
36	pPathHandleRet	4	
40	sPathHandleRet	2	2

This page intentionally left blank

ScsiQueryInfo (wModule wBus, wFmt, pScsiInfoRet, sScsiInfoMax, pCbScsiInfoRet): ercType

Description

ScsiQueryInfo scans the specified SCSI bus and returns information about the devices connected to it. With each invocation of ScsiQueryInfo, the operation returns information about the devices currently powered on.

The information can be returned in either of two formats. The short format returns 3 bytes: the target ID, the LUN, and the peripheral device type. The long format returns this information and 35 bytes of more detailed information about the device. Both formats are described below.

ScsiQueryInfo limits its examination to LUN 0 of each target. To obtain information about other logical units associated with a target, the application can build an INQUIRY command descriptor block and use the operation ScsiCdbDataIn to issue the request to the desired SCSI target(s) and logical unit(s).

Because the information returned in the fields *deviceType* through *productRevision* is obtained directly from the SCSI device, the precise meanings of these fields may vary with the device queried. Use the format descriptions shown below as a general guide to field contents. For details, consult the SCSI standard and the manufacturer's manual for each device attached to the SCSI bus.

ScsiQueryInfo is another name for the GetScsiInfo operation. See Appendix J, "Operation Data," to determine which operating system versions support ScsiQueryInfo.

Each short format entry is as follows:

Offset	Field	Size (Bytes)	Contents
0	targetID	1	SCSI device ID.
1	lun	1	SCSI logical unit number.
2	deviceType	1	Peripheral device type. (See the description of *deviceType* following the formats.)

Each long format entry is as follows:

Offset	Field	Size (Bytes)	Contents
0	targetID	1	SCSI device ID.
1	LUN	1	SCSI logical unit number.
2	deviceType	1	Peripheral device type. (See the description of *deviceType* following this format.)
3	removableMedium	1	The most significant bit is set if the device medium is removable.
4	version	1	Bits in this field describe the ISO, ECMA, and ANSI standard levels supported.
5	responseDataFormat	1	
6	additionalLength	1	Count of additional bytes of data that follow. ScsiQueryInfo truncates inquiry data available from the device that extends beyond the first 36 bytes.
7	reserved	2	
9	deviceCapabilities	1	Bits in this field describe device-specific capabilities.

Offset	Field	Size (Bytes)	Contents
10	vendorID	8	Vendor ID of the device (left justified ASCII text).
18	productID	16	Product ID of the device (left justified ASCII text).
34	productRevision	4	Product revision level of the device (left justified ASCII text).

The field *deviceType* can be any value defined in the SCSI standard. The most significant 3 bits are a device type qualifier that modifies the least significant 5 bits, a number that represents the device type. Examples of some common values are described below:

Value	Description
0	Direct-access device (for example, magnetic disk)
1	Sequential-access device (for example, magnetic tape)
127	Logical unit is not present

Procedural Interface

ScsiQueryInfo (wModule wBus, wFmt, pScsiInfoRet, sScsiInfoMax, pCbScsiInfoRet): ercType

where

wModule

is a word (on workstations) indicating the X-Bus position of the module controlling the SCSI bus. The value is 0 for the processor module, 1 for the next module to the right, and so forth. From this value, the specified SCSI bus can be derived. On shared resource processors, *wModule* is ignored because ScsiQueryInfo returns information only for the processor board from which the request was made.

wBus

is a word (on shared resource processors) specifying the SCSI bus number. On workstations, *wBus* is ignored (the SCSI bus number is derived from the value of *wModule*).

wFmt

is a word that specifies the format of the returned information. The allowed values are as follows:

Value	**Description**
0	Returns the short format
1	Returns the long format

pScsiInfoRet
sScsiInfoMax

describe the memory area into which the information requested is to be returned.

pCbScsiInfoRet

is the memory address of a word into which the count of information bytes is placed. The value returned is 0 if the specified module (workstation) or processor (shared resource processor) does not control a SCSI bus, the requested SCSI bus does not exist, or no powered-on devices are connected to the SCSI bus.

Request Block

sCbScsInfoRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	353
12	wModule	2	
14	wBus	2	
16	wFmt	2	
18	pScsiInfoRet	4	
22	sScsiInfoMax	2	
24	pCbScsiInfoRet	4	
28	sCbScsiInfoRet	2	2

This page intentionally left blank

ScsiQueryPathParameters (pathHandle, pPathParametersRet, sPathParametersMax): ercType

Description

ScsiQueryPathParameters returns the virtual path characteristics for SCSI communications between the host adapter and SCSI target and LUN specified by the SCSI path handle.

ScsiQueryPathParameters does not return any information for paths opened in SCSI Manager target mode. Status code 385 ("Invalid SCSI path handle") is returned in this case.

Procedural Interface

ScsiQueryPathParameters (pathHandle, pPathParametersRet, sPathParametersMax) :ercType

where

pathHandle

is the SCSI path handle (word) returned by a ScsiOpenPath operation.

pPathParametersRet
sPathParametersMax

describe an area for the return of a path parameters block. The format of this structure is shown below:

Offset	**Field**	**Size (Bytes)**
0	fNoDisconnect	1
1	bSynchronousOffset	1
2	wSynchronousPeriod	2
4	wDefaultTimeout	2
6	reserved	10

See the description of ScsiOpenPath for the meanings of the fields in the path parameters block.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	356
12	pathHandle	2	
14	reserved	4	
18	pPathParametersRet	4	
22	sPathParametersMax	2	

This page intentionally left blank

ScsiRequestSense (pathHandle, wTimeout, pExtSenseRet, sExtSenseMax, pCbExtSenseRet, pStatusRet): ercType

Caution: It is not essential that this operation be used. Other SCSI operations for sending commands to the SCSI target (for example, ScsiCdbDataIn and ScsiCdbDataIn) include a buffer for saving sense information that may be returned. This operation should only be used if the SCSI path is opened in exclusive mode. Otherwise, the sense data returned may not be meaningful.

Description

ScsiRequestSense requests extended sense information from the SCSI target and LUN indicated by the path handle.

Because SCSI devices clear sense data upon receipt of a new command if the command is not REQUEST SENSE, use of the ScsiRequestSense operation is not recommended unless the caller is the sole user of the SCSI device. If a buffer for the return of sense data is provided in the ScsiCdbDataIn or ScsiCdbDataOut operations, the SCSI Manager automatically obtains sense information when it has been created by a SCSI device CHECK CONDITION.

ScsiRequestSense may not be used with paths opened in SCSI Manager target mode; status code 385 ("Invalid SCSI path handle") is returned in this case.

Procedural Interface

ScsiRequestSense (pathHandle, wTimeout, pExtSenseRet, sExtSenseMax, pCbExtSenseRet, pStatusRet): ercType

where

pathHandle

is the SCSI path handle (word) returned by a ScsiOpenPath operation.

wTimeout

is a word that specifies the timeout value (in tenths of a second) for the operation. Zero indicates that the default timeout for the SCSI path should be used.

pExtSenseRet
sExtSenseMax

describe an area where extended sense from the target is returned. This information is valid only if ercOK is returned by ScsiRequestSense and the target status returned is GOOD.

pCbExtSenseRet

is the address of a word to which the count of bytes of extended sense information is returned.

pStatusRet

is the address of a byte to which status from the target is returned. (The status byte values are contained in the ANSI SCSI standard.)

Request Block

sCbExtSenseRet is always 2 and *sStatusRet* is always 1.

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	2	6
2	nReqPbCb	1	1
3	nRespPbCb	1	4
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	362
12	pathHandle	2	
14	reserved	2	
16	wTimeout	2	
18	reserved	6	
24	pExtSenseRet	4	
28	sExtSenseMax	2	
30	pCbExtSenseRet	4	
34	sCbExtSenseRet	2	2
36	pStatusRet	4	
40	sStatusRet	2	1
42	reserved	6	

This page intentionally left blank

ScsiReset (pathHandle, fBusReset): ercType

Description

ScsiReset forces a reset (asserts the RST signal) of either all devices on the SCSI bus or one device only and aborts any commands in progress. If a reset of the entire bus is generated, this may affect other users of the SCSI Manager or other host adapters connected to the SCSI bus.

ScsiReset may not be used with paths opened in SCSI Manager target mode. Status code 385 ("Invalid SCSI path handle") is returned in this case.

Procedural Interface

ScsiReset (pathHandle, fBusReset): ercType

where

pathHandle

is the SCSI path handle (word) returned by a ScsiOpenPath operation. Note that the target ID and LUN specified in a ScsiOpenPath operation are immaterial for obtaining a path handle to reset a host adapter (*fBusReset* is TRUE; see below) and all the SCSI targets it controls on the SCSI bus. Only the host adapter portion of the path specification is relevant in this case.

fBusReset

is a flag that is TRUE if the RST signal is to be asserted for all devices on the SCSI bus. The flag is FALSE if a BUS DEVICE RESET message is to be sent to a specified peripheral. Note that BUS DEVICE RESET must be supported by the SCSI Manager, or status code 7 ("Not implemented") is returned. A value of TRUE for *fBusReset* is supported by all SCSI Manager versions.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	2	4
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	359
12	pathHandle	2	
14	fBusReset	1	
15	reserved	1	

ScsiSetPathParameters (pathHandle, pPathParameters, sPathParameters): ercType

Description

ScsiSetPathParameters changes the virtual path characteristics for SCSI communications between the host adapter and SCSI target and LUN. The path parameters remain in effect until they are altered by a subsequent call to ScsiSetPathParameters.

ScsiSetPathParameters may not be used with paths opened in SCSI Manager target mode. Status code 385 ("Invalid SCSI path handle") is returned in this case.

Procedural Interface

ScsiSetPathParameters (pathHandle, pPathParameters, sPathParameters): ercType

where

pathHandle

is the SCSI path handle (word) returned by a ScsiOpenPath operation.

pPathParameters
sPathParameters

describe an area for the return of a path parameters block. The format of this structure is shown below:

Offset	**Field**	**Size (Bytes)**
0	fNoDisconnect	1
1	bSynchronousOffset	1
2	wSynchronousPeriod	2
4	wDefaultTimeout	2
6	reserved	10

See the description of ScsiOpenPath for the meanings of the fields in the path parameters block.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	2	6
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	357
12	pathHandle	2	
14	reserved	4	
18	pPathParameters	4	
22	sPathParameters	2	

This page intentionally left blank

ScsiTargetCdbCheck (pathHandle, pBusIdRet, pCdbRet, sCdbMax): ercType

Description

ScsiTargetCdbCheck determines whether a SCSI SEND or RECEIVE command from another SCSI initiator has been received by the LUN of the local processor device specified by the path handle.

If a command has been received, it is currently in a suspended (disconnected) state from the viewpoint of the SCSI initiator that sent the command. In this case, the command descriptor block returned by ScsiTargetCdbCheck may be used to determine the appropriate buffer size to allocate for a ScsiTargetDataReceive or ScsiTargetDataTransmit operation to complete the SEND or RECEIVE command.

If no command has been received, status code 387 ("No CDB available") is returned.

For paths not opened in SCSI Manager target mode, ScsiTargetCdbCheck returns status code 385 ("Invalid SCSI path handle").

Procedural Interface

ScsiTargetCdbCheck (pathHandle, pBusIdRet, pCdbRet, sCdbMax): ercType

where

pathHandle

is the SCSI path handle (word) returned by a ScsiOpenPath operation. The path must have been opened for target mode operations.

pBusIdRet

is the memory address of a byte to which the SCSI bus identification number (ID) of the initiator that issued the SEND or RECEIVE command is returned.

pCdbRet
sCdbMax

describe an area for the return of the command descriptor block transmitted by the SCSI initiator. The area must be large enough to accomodate the block, or status code 39 ("Return data area too small") is returned.

Request Block

sBusIdRet is always 1.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	389
12	pathHandle	2	
14	reserved	4	
18	pBusIdRet	4	
22	sBusIdRet	2	1
24	pCdbRet	4	
28	sCdbMax	2	

ScsiTargetCdbWait (pathHandle, pBusIdRet, pCdbRet, sCdbMax): ercType

Description

ScsiTargetCdbWait determines whether a SCSI SEND or RECEIVE command from another SCSI initiator has been received by the LUN of the local processor device specified by the path handle.

If a command has been received, it is currently in a suspended (disconnected) state from the viewpoint of the SCSI initiator that sent the command. In this case, the command descriptor block returned by ScsiTargetCdbWait may be used to determine the appropriate buffer size to allocate for a ScsiTargetDataReceive or ScsiTargetDataTransmit operation to complete the SEND or RECEIVE command.

If no command has been received, the requesting process is suspended until a command is received from a SCSI initiator.

For paths not opened in SCSI Manager target mode, ScsiTargetCdbWait returns status code 385 ("Invalid SCSI path handle").

Procedural Interface

ScsiTargetCdbWait (pathHandle, pBusIdRet, pCdbRet, sCdbMax): ercType

where

pathHandle

is the SCSI path handle (word) returned by a ScsiOpenPath operation. The path must have been opened for target mode operations.

pBusIdRet

is the memory address of a byte to which the SCSI bus identification number (ID) of the initiator that issued the SEND or RECEIVE command is returned.

pCdbRet
sCdbMax

describe an area for the return of the command descriptor block transmitted by the SCSI initiator. The area must be large enough to accomodate the block, or status code 39 ("Return data area too small") is returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	390
12	pathHandle	2	
14	reserved	4	
18	pBusIdRet	4	
22	sBusIdRet	2	1
24	pCdbRet	4	
28	sCdbMax	2	

ScsiTargetDataReceive (pathHandle, pCdbRet, sCdbMax, pDataRet, sDataMax, pCbDataRet, pStatusRet, pExtSenseRet, sExtSenseMax): ercType

Description

ScsiTargetDataReceive enables the SCSI Manager to accept a SCSI SEND command from another initiator on the SCSI bus and to receive up to the maximum amount of data specified by the operation during a DATA OUT phase associated with the SEND command. The command descriptor block issued by the other initiator, the count of data bytes received, and the SCSI status transmitted to the other initiator are all returned by this operation.

For paths not opened in SCSI Manager target mode, ScsiTargetDataReceive returns status code 385 ("Invalid SCSI path handle").

Procedural Interface

ScsiTargetDataReceive (pathHandle, pCdbRet, sCdbMax, pDataRet, sDataMax, pCbDataRet, pStatusRet, pExtSenseRet, sExtSenseMax): ercType

where

pathHandle

is the SCSI path handle (word) returned by a ScsiOpenPath operation. The path must have been opened for target mode operations.

pCdbRet
sCdbMax

describe an area for the return of the command descriptor block transmitted by the SCSI initiator. The area must be large enough to accomodate the block, or status code 39 ("Return data area too small") is returned.

pDataRet
sDataMax

describe the buffer area available to receive information sent from the SCSI initiator during the DATA OUT phase.

pCbDataRet

is the address of a word to which the actual count of bytes of data transferred from the SCSI initiator during the DATA OUT phase is returned.

pStatusRet

is the address of a byte to which the status sent to the SCSI initiator by the SCSI Manager is returned. (The status byte values are contained in the ANSI SCSI standard.)

pExtSenseRet
sExtSenseMax

describe an area where extended sense sent to the SCSI initiator by the SCSI Manager is returned if the SEND command from the SCSI initiator is terminated with a target status of CHECK CONDITION. The sense data information is valid only if status code 0 ("ercOK") is returned by ScsiTargetDataReceive. If *pExtSenseRet* is 0 or *sExtSenseMax* is 0, the SCSI Manager does not report the sense data sent to the SCSI initiator.

Request Block

sCbDataRet is always 2 and *sStatusRet* is always 1.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	5
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	387
12	pathHandle	2	
14	reserved	4	
18	pDataRet	4	
22	sDataMax	2	
24	pCdbRet	4	
28	sCdbMax	2	
30	pCbDataRet	4	
34	sCbDataRet	2	2
36	pStatusRet	4	
40	sStatusRet	2	1
42	pExtSenseRet	4	
44	sExtSenseMax	2	

This page intentionally left blank

ScsiTargetDataReceiveAsync (pathHandle, pCdbRet, sCdbMax, pDataRet, sDataMax, pExtSenseRet, sExtSenseMax, pRq, exchangeReply): ercType

Description

ScsiTargetDataReceiveAsync enables the SCSI Manager to accept a SCSI SEND command from another initiator on the SCSI bus and to receive up to the maximum amount of data specified by the operation during a DATA OUT phase associated with the SEND command. ScsiWaitTargetDataAsync must be called to check the completion status of the operation.

ScsiTargetDataReceiveAsync is a library procedure that provides an interface to the ScsiTargetDataReceive request. (See ScsiTargetDataReceive.) The information returned by the ScsiTargetDataReceive arguments *pCbDataRet* and *pStatusRet* is not available when the SCSI operation is initiated but is returned when the ScsiWaitTargetDataAsync operation is performed.

Procedural Interface

ScsiTargetDataReceiveAsync (pathHandle, pCdbRet, sCdbMax, pDataRet, sDataMax, pExtSenseRet, sExtSenseMax, pRq, exchangeReply): ercType

where

pathHandle

is the SCSI path handle (word) returned by a ScsiOpenPath operation. The path must have been opened for target mode operations.

pCdbRet
sCdbMax

describe an area for the return of the command descriptor block transmitted by the SCSI initiator. The area must be large enough to accomodate the block, or status code 39 ("Return data area too small") is returned.

pDataRet
sDataMax

describe the buffer area available to receive information sent from the SCSI initiator during the DATA OUT phase.

pExtSenseRet
sExtSenseMax

describe an area where extended sense sent to the SCSI initiator by the SCSI Manager is returned if a subsequent call to ScsiWaitTargetDataAsync indicates the SEND command from the SCSI initiator was terminated with a target status of CHECK CONDITION. The sense data information is valid only if status code 0 ("ercOK") is returned by ScsiWaitTargetDataAsync. If *pExtSenseRet* is 0 or *sExtSenseMax* is 0, the SCSI Manager does not report the sense data sent to the SCSI initiator.

pRq

is the address of a 64-byte area to be used as a work space by the ScsiTargetDataReceiveAsync and ScsiWaitTargetDataAsync operations.

exchangeReply

is an exchange (word) provided by the user for the exclusive use of ScsiTargetDataReceiveAsync and ScsiWaitTargetDataAsync.

Request Block

ScsiTargetDataReceiveAsync is an object module procedure that provides an interface to the ScsiTargetDataReceive request (see ScsiTargetDataReceive).

ScsiTargetDataTransmit (pathHandle, pCdbRet, sCdbMax, pData, sData, pCbDataRet, pStatusRet, pExtSenseRet, sExtSenseMax): ercType

Description

ScsiTargetDataTransmit enables the SCSI Manager to accept a SCSI RECEIVE command from another initiator on the SCSI bus and to transmit up to the maximum amount of data specified by the operation during a DATA IN phase associated with the RECEIVE command. The command descriptor block issued by the other initiator, the count of data bytes transmitted, and the SCSI status transmitted to the other initiator are all returned by this operation.

For paths not opened in SCSI Manager target mode, ScsiTargetDataTransmit returns status code 385 ("Invalid SCSI path handle").

Procedural Interface

ScsiTargetDataTransmit (pathHandle, pCdbRet, sCdbMax, pData, sData, pCbDataRet, pStatusRet, pExtSenseRet, sExtSenseMax): ercType

where

pathHandle

is the SCSI path handle (word) returned by a ScsiOpenPath operation. The path must have been opened for target mode operations.

pCdbRet
sCdbMax

describe an area for the return of the command descriptor block transmitted by the SCSI initiator. The area must be large enough to accomodate the block, or status code 39 ("Return data area too small") is returned.

pData
sData

describe the buffer area from which to transmit information to the SCSI initiator during the DATA IN phase.

pCbDataRet

is the address of a word to which the actual count of bytes of data transferred from the SCSI device during the DATA IN phase is returned.

pStatusRet

is the address of a byte to which the status sent to the SCSI initiator by the SCSI Manager is returned. (The status byte values are contained in the ANSI SCSI standard.)

pExtSenseRet
sExtSenseMax

describe an area where extended sense data sent to the SCSI initiator by the SCSI Manager is returned if the RECEIVE command from the SCSI initiator is terminated with a target status of CHECK CONDITION. The sense data information is valid only if status code 0 ("ercOK") is returned by ScsiTargetDataTransmit. If *pExtSenseRet* is 0 or *sExtSenseMax* is 0, the SCSI Manager does not report the sense data sent to the SCSI initiator.

Request Block

sCbDataRet is always 2 and *sStatusRet* is always 1.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	1
3	nRespPbCb	1	4
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	388
12	pathHandle	2	
14	reserved	4	
18	pData	4	
22	sData	2	
24	pCdbRet	4	
28	sCdbMax	2	
30	pCbDataRet	4	
34	sCbDataRet	2	2
36	pStatusRet	4	
40	sStatusRet	2	1
42	pExtSenseRet	4	
44	sExtSenseMax	2	

This page intentionally left blank

ScsiTargetDataTransmitAsync (pathHandle, pCdbRet, sCdbMax, pData, sData, pExtSenseRet, sExtSenseMax, pRq, exchangeReply): ercType

Description

ScsiTargetDataTransmitAsync enables the SCSI Manager to accept a SCSI RECEIVE command from another initiator on the SCSI bus and to transmit up to the maximum amount of data specified by the operation during a DATA IN phase associated with the RECEIVE command. ScsiWaitTargetDataAsync must be called to check the completion status of the operation.

ScsiTargetDataTransmitAsync is a library procedure that provides an interface to the ScsiTargetDataTransmit request. (See ScsiTargetDataTransmit.) The information returned by the ScsiTargetDataTransmit arguments *pCbDataRet* and *pStatusRet* is not available when the SCSI operation is initiated but is returned when the ScsiWaitTargetDataAsync operation is performed.

Procedural Interface

ScsiCdbDataInAsync (pathHandle, pCdbRet, cCdbMax, pData, sData, pExtSenseRet, sExtSenseMax, pRq, exchangeReply): ercType

where

pathHandle

is the SCSI path handle (word) returned by a ScsiOpenPath operation. The path must have been opened for target mode operations.

pCdbRet
sCdbMax

describe an area for the return of the command descriptor block transmitted by the SCSI initiator. The area must be large enough to accomodate the block, or status code 39 ("Return data area too small") is returned.

pData
sData

describe the buffer area available to transmit information to the SCSI initiator during the DATA IN phase.

pExtSenseRet
sExtSenseMax

describe an area where extended sense sent to the SCSI initiator by the SCSI Manager is returned if a subsequent call to ScsiWaitTargetDataAsync indicates the SEND command from the SCSI initiator was terminated with a target status of CHECK CONDITION. The sense data information is valid only if status code 0 ("ercOK") is returned by ScsiWaitTargetDataAsync. If *pExtSenseRet* is 0 or *sExtSenseMax* is 0, the SCSI Manager does not report the sense data sent to the SCSI initiator.

pRq

is the address of a 64-byte area to be used as a work space by the ScsiTargetDataTransmitAsync and ScsiWaitTargetDataAsync operations.

exchangeReply

is an exchange (word) provided by the user for the exclusive use of ScsiTargetDataTransmitAsync and ScsiWaitTargetDataAsync.

Request Block

ScsiTargetDataTransmitAsync is an object module procedure that provides an interface to the ScsiTargetDataTransmit request (see ScsiTargetDataTransmit).

ScsiTargetOperationsAbort(pathHandle): ercType

Description

ScsiTargetOperationsAbort cancels any pending ScsiTargetDataReceive, ScsiTargetDataTransmit, or ScsiTargetCdbWait operation that is awaiting selection by a SCSI initiator. This operation is useful when a ScsiClosePath is to be issued but there are still outstanding SCSI target mode requests. If a SCSI initiator has already commenced selection of a previously pending operation, status code 394 ("SCSI requests outstanding") is returned. In this case, the caller must wait for the completion of the SCSI target mode operation(s) before issuing the ScsiClosePath.

Procedural Interface

ScsiTargetOperationsAbort(pathHandle): ercType

where

pathHandle

is the SCSI path handle (word) returned by a ScsiOpenPath operation in target processor mode.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	2
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	405
12	pathHandle	2	

ScsiWaitCdbAsync (pRq, pCbDataRet, pStatusRet): ercType

Description

ScsiWaitCdbAsync synchronizes a process with an asynchronous SCSI operation initiated by ScsiCdbDataInAsync or ScsiCdbDataOutAsync. For example, after calling ScsiCdbDataInAsync with a SCSI READ command, the SCSI command descriptor block is queued at the SCSI Manager and the requesting process continues execution. When the process needs to synchronize with the asynchronous SCSI operation (that is, await its completion), the process calls ScsiWaitCdbAsync. ScsiWaitCdbAsync waits for the operation to complete, checks the status code, and obtains both the byte count of data transferred and the target status.

For paths opened in SCSI Manager target mode, ScsiWaitCdbAsync returns status code 385 ("Invalid SCSI path handle"). Status code 248 ("Wrong pRq argument") is returned if the *pRq* argument does not match the argument of a previous ScsiCdbDataInAsync or ScsiCdbDataOutAsync.

Procedural Interface

ScsiWaitCdbAsync (pRq, pCbDataRet, pStatusRet): ercType

where

pRq

is the address of the 64-byte work area previously supplied in the *pRq* argument of a ScsiCdbDataInAsync or ScsiCdbDataOutAsync.

pCbDataRet

is the address of a word to which the actual count of bytes of data transferred to or from the SCSI device during the DATA OUT or DATA IN phase is returned.

pStatusRet

is the address of a byte to which status from the target is returned. (The status byte values are contained in the ANSI SCSI standard.)

Request Block

ScsiWaitCdbAsync is an object module procedure that provides an interface to the ScsiCdbDataIn and ScsiCdbDataOut requests. (See ScsiCdbDataIn and ScsiCdbDataOut.)

ScsiWaitTargetDataAsync (pRq, pCbDataRet, pStatusRet): ercType

Description

ScsiWaitTargetDataAsync synchronizes a process with an asynchronous SCSI operation initiated by ScsiTargetDataReceiveAsync or ScsiTargetDataTransmitAsync. For example, after calling ScsiTargetDataReceiveAsync, the request is queued at the SCSI Manager and the requesting process continues execution. When the process needs to synchronize with the asynchronous SCSI operation (that is, await its completion), the process calls ScsiWaitTargetDataAsync. ScsiWaitTargetDataAsync waits for the operation to complete, checks the status code, and obtains both the byte count of data transferred and the target status.

Status code 248 ("Wrong pRq argument") is returned if the *pRq* argument does not match the argument of a previous ScsiTargetDataReceiveAsync or ScsiTargetDataTransmitAsync.

NOTE: There is no timeout requirement for this operation. The call to ScsiTargetDataTransmitAsync or ScsiTargetDataReceiveAsync establishes a data buffer to transmit data to or receive data from another initiator on the SCSI bus. If the initiator never issues a command to receive or send data, ScsiWaitTargetDataAsync will wait forever.

Procedural Interface

ScsiWaitTargetDataAsync (pRq, pCbDataRet, pStatusRet): ercType

where

pRq

is the address of the 64-byte work area previously supplied in the *pRq* argument of a ScsiTargetDataReceiveAsync or ScsiTargetDataTransmitAsync operation.

pCbDataRet

is the address of a word to which the actual count of bytes of data transferred to or from the SCSI initiator during the DATA IN or DATA OUT phase is returned.

pStatusRet

is the address of a byte to which status sent to the SCSI initiator is returned. (The status byte values are contained in the ANSI SCSI standard.)

Request Block

ScsiWaitTargetDataAsync is an object module procedure that provides an interface to the ScsiTargetDataReceive and ScsiTargetDataTransmit requests. (See ScsiTargetDataReceive and ScsiTargetDataTransmit.)

SemClear (sh): ercType

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

SemClear performs one of two functions on the semaphore specified by *sh*. If the semaphore is a signaling semaphore, it clears the semaphore set by the SemSet operation. If the semaphore is a mutual exclusion lock semaphore, it decrements the count and clears it when the count is zero.

SemClear clears the specified signaling semaphore regardless of the semaphore state. This action notifies all waiting processes that the semaphore has been cleared. Such processes wait for the signaling semaphore with one of the following signaling semaphore operations: SemWait, SemMuxWait, or SemNotify. (See the descriptions of these operations for details.)

SemClear is paired with the SemLock operation to clear the specified mutual exclusion semaphore. (See SemLock.) If the semaphore is locked more than once by the same process as can occur when SemLock is called repeatedly in a nested procedure, the call to SemClear merely decrements the lock count associated with the semaphore. Only when the lock count reaches 0 does SemClear actually clear the lock.

The SemClear operation is similar to SemClearCritical. The latter operation, however, should only be used with the SemLockCritical operation to lock and clear a critical section semaphore. (See SemClearCritical.)

If the caller tries to clear a semaphore opened for exclusive use by a different user number, status code 13954 ("Invalid semaphore user") is returned.

For details on semaphores, see the section entitled "Semaphores" in the *CTOS Operating System Concepts Manual.*

Procedural Interface

SemClear (sh): ercType

where

sh

is the 32-bit semaphore handle.

Request Block

SemClear is a system-common procedure.

SemClearCritical (sh): ercType

Caution: *Exercise care when providing the semaphore handle to this operation. If the handle is invalid, a status code is not returned. An invalid handle may cause data to be overwritten.*

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

SemClearCritical clears the critical section semaphore specified by *sh* that was previously locked by the SemLockCritical operation.

The SemClearCritical operation is similar to SemClear. The latter operation, however, should only be used with the SemLock operation to lock and clear a noncritical semaphore. (See SemClear.)

For details on semaphores, see the section entitled "Semaphores" in the *CTOS Operating System Concepts Manual.*

Procedural Interface

SemClearCritical (sh): ercType

where

sh

is the 32-bit semaphore handle.

Request Block

SemClearCritical is a system-common procedure.

SemClearProcessLocks (pid): ercType

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases.*

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

SemClearProcessLocks clears all the terminatable system semaphores locked by the specified process. This operation is a subroutine of the termination process.

Procedural Interface

SemClearProcessLocks (pid): ercType

where

pid

is the process identifier.

Request Block

SemClearProcessLocks is a system-common procedure.

This page intentionally left blank.

SemClose (sh): ercType

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

SemClose closes the system semaphore specified by *sh* that was previously opened by the SemOpen operation. When the semaphore is closed by all its users, it is deallocated.

If the caller attempts to close a locked semaphore, status code 13958 ("Semaphore locked") is returned.

For details on semaphores, see the section entitled "Semaphores" in the *CTOS Operating System Concepts Manual.*

Procedural Interface

SemClose (sh): ercType

where

sh

is the 32-bit semaphore handle.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	407
12	sh	4	

SemEnumerate (userNum, index, pShRet): ercType

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases.*

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

SemEnumerate returns the semaphore handle of the system semaphore opened by the specified user number.

The caller specifies the handle with the *index* parameter. Index values start with the value 0. A list of all the open semaphores for a given user number can be obtained by calling SemEnumerate repeatedly, incrementing the value of *index* in each call until status code 13953 ("Invalid semaphore handle") is returned.

The Debugger uses this operation to return semaphore information when the end user types the **Code-Y** command. (See the *CTOS Debugger User's Guide* for details.)

Procedural Interface

SemEnumerate (userNum, index, pShRet): ercType

where

userNum

is the user number opening the semaphore.

index

specifies which semaphore handle to return.

pShRet

is the memory address of a double word where the semaphore handle is returned.

Request Block

SemEnumerate is a system-common procedure.

SemEnumerateWaiting (sh, index, pbSemWaitRecordRet, cbSemWaitRecordRet): ercType

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases.*

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

SemEnumerateWaiting returns the waiting record for the system semaphore specified by *sh*.

The caller specifies the waiting record with the *index* parameter. Index values start with the value 0. A list of all the waiting records can be obtained by calling SemEnumerateWaiting repeatedly, incrementing the value of *index* in each call until status code 13953 ("Invalid semaphore handle") is returned.

The Debugger uses this operation to return semaphore information when the end user types the **Code-Y** command. (See the *CTOS Debugger User's Guide* for details.) Note that critical section semaphores do not have waiting records. The only way the Debugger can determine if there are any waiting processes is to see if bit 5 is set in the *status* field of the Process Control Block (PCB) of each process on the run queue.

Procedural Interface

SemEnumerateWaiting (sh, index, pbSemWaitRecordRet, cbSemWaitRecordRet): ercType

where

sh

is the 32-bit semaphore handle.

index

specifies the waiting record to be returned.

pbSemWaitRecordRet
cbSemWaitRecordRet

describe the waiting record. The record has the following format:

Offset	Field	Size (Bytes)	Description
0	pNext	4	The memory address of the next waiting record in the list of waiting records.
4	pPrev	4	The memory address of the previous waiting record.
8	tss	2	The Task State Segment (TSS) of the waiting process.
10	priority	2	The priority of the waiting process.
12	exch	2	The exchange to which a message is sent when the semaphore is cleared, or when a timeout occurs.

Offset	Field	Size	Description
14	timeout	2	The amount of time the process will wait for the semaphore to clear before a timeout occurs. **(Bytes)**
16	flag	2	Indicates whether or not the process is waiting using the SemNotify operation. The process is using SemNotify if the low bit is 1.

Request Block

SemEnumerateWaiting is a system-common procedure.

This page intentionally left blank.

SemLock (sh, timeout): ercType

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

SemLock causes the caller to wait until the mutual exclusion semaphore specified by *sh* is cleared. Then it sets the lock.

If the mutual exclusion semaphore is a system semaphore the calling process already owns, SemLock increments the lock count, and the caller continues processing as if it just acquired the lock.

The SemLock operation is similar to SemLockCritical. The latter operation, however, should only be used with the SemClearCritical operation to lock and clear a critical section semaphore. (See SemLockCritical.)

Status code 13956 ("Semaphore owner killed") is returned if the process that locked the semaphore is terminated. If the use count reaches FFFFh, status code 13957 ("Semaphore count overflow") is returned.

For details on semaphores, see the section entitled "Semaphores" in the *CTOS Operating System Concepts Manual.*

Procedural Interface

SemLock (sh, timeout): ercType

where

sh

is the 32-bit semaphore handle.

timeout

specifies the amount of time the process will wait for the semaphore to be cleared. The timeout values and their meanings are

Value	**Meaning**
0	Do not wait. If the semaphore is not clear, return status code 13955 ("Semaphore timeout").
-1	Wait indefinitely for the semaphore to be cleared.
Greater than 0	Wait for the specified amount of time in units of 100 milliseconds. If the semaphore still is not clear, return status code 13955 ("Semaphore timeout").

Request Block

SemLock is a system-common procedure.

SemLockCritical (sh): ercType

Caution: *Exercise care when providing the semaphore handle to this operation. If the handle is invalid, a status code is not returned. An invalid handle may overwrite code.*

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

SemLockCritical causes the caller to wait until the critical section semaphore specified by *sh* is cleared. Then it sets the lock.

The SemLockCritical operation is similar to SemLock. The latter operation, however, should only be used with the SemClear operation to lock and clear a noncritical semaphore. (See SemLock.)

For details on semaphores, see the section entitled "Semaphores" in the *CTOS Operating System Concepts Manual.*

Procedural Interface

SemLockCritical (sh): ercType

where

sh

is the 32-bit semaphore handle.

Request Block

SemLockCritical is a system-common procedure.

SemMuxWait (pIndex, pSemaphoreList, timeout): ercType

NOTES:

1. *This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.*
2. *The SemMuxWait, SemNotify, SemSet, and SemWait operations should be used for signaling, not for mutual exclusion.*

Description

SemMuxWait allows the calling process to wait for more than one signaling semaphore previously set by the SemSet operation. If one of the semaphores is already clear, no waiting is necessary. Otherwise the caller waits at its default response exchange until one of the semaphores it is waiting for clears, or a timeout occurs. If more than one semaphore clears while the caller is waiting, SemMuxWait returns the index of the first clear semaphore in the list.

SemMuxWait differs from the SemWait operation in that the latter only allows the caller to wait for one semaphore. (See SemWait.)

For details on semaphores, see the section entitled "Semaphores" in the *CTOS Operating System Concepts Manual.*

Procedural Interface

SemMuxWait (pIndex, pSemaphoreList, timeOut): ercType

where

pIndex

is the memory address of a word where the index of the cleared semaphore is returned.

pSemaphoreList

is the memory address of the list of signaling semaphores the caller is waiting for. The first word in the list is the semaphore count. It specifies the number of semaphores the caller is waiting for. The semaphore count is followed by a series of 6 byte entries, one for each semaphore. The format of each entry is shown below:

Field	**Size**	**Description**
rsvd	2	Reserved.
sh	4	The 32-bit semaphore handle.

timeout

specifies the amount of time the process will wait for the semaphore to be cleared. The timeout values and their meanings are

Value	**Meaning**
0	Do not wait. If all semaphores in the list are set, status code 13955 ("Semaphore timeout") is returned.
-1	Wait indefinitely until one of the semaphores is clear.
Greater than 0	Wait for the specified amount of time in units of 100 milliseconds. If the semaphore still is not clear, return status code 13955 ("Semaphore timeout").

Request Block

SemMuxWait is a system-common procedure.

This page intentionally left blank.

SemNotify (sh, timeout, exch): ercType

NOTES:

1. *This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.*
2. *The SemMuxWait, SemNotify, SemSet, and SemWait operations should be used for signaling, not for mutual exclusion.*

Description

SemNotify sends a message to the exchange provided to notify the caller that the system semaphore specified by *sh* is clear. If the semaphore is already clear, SemNotify sends the message immediately. Otherwise, it waits. If a timeout occurs before the semaphore is clear, SemNotify sends the message, and control returns to the caller immediately.

The message sent to the specified exchange is the semaphore handle. The caller should view this message as a hint only that the semaphore is clear. By the time the message is received, a timeout already could have occurred, or the semaphore could have been set again by another process. To determine if the semaphore indeed is still clear, the caller should call the SemWait operation with a 0 value for the timeout parameter. (See SemWait.)

For details on semaphores, see the section entitled "Semaphores" in the *CTOS Operating System Concepts Manual.*

Procedural Interface

SemNotify (sh, timeout, exch): ercType

where

sh

is the 32-bit semaphore handle.

timeout

is the amount of time during which the process should wait for the semaphore to be cleared. Each value indicates the following:

Value	Meaning
0	Do not wait. Send a message to the specified exchange immediately.
-1	Wait indefinitely for the semaphore to be cleared.
Greater than 0	Wait for the specified amount of time in units of 100 milliseconds. If the semaphore still is not clear, return status code 13955 ("Semaphore timeout").

exch

is the exchange where the message is sent.

Request Block

SemNotify is a system-common procedure.

SemOpen (pbSemName, cbSemName, options, pShRet): ercType

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

SemOpen opens a system semaphore and returns a semaphore handle. If no semaphore exists for the specified name or if no name is specified, a new system semaphore is created. If the semaphore is already open, SemOpen returns the semaphore handle originally assigned.

Multiple users may open the same semaphore. To share a system semaphore, two or more processes must first agree on the semaphore name and must specify the same semaphore characteristics for the *options* parameter as established in the initial call to SemOpen.

If the caller tries to clear a semaphore opened for exclusive use by a different user number, status code 13954 ("Invalid semaphore user") is returned. Status code 13951 ("Semphore name too long") is returned if the semaphore name is longer than 36 bytes.

If a semaphore cannot be created because the memory available is inadequate, status code 13950 ("No semaphore memory") is returned. You can specify the amount of memory to allocate for system semaphores through a system configuration option. (For details, see "Configurable Parameters" in Section 16 of the *CTOS System Administration Guide*.)

For details on semaphores, see the section entitled "Semaphores" in the *CTOS Operating System Concepts Manual*.

Procedural Interface

SemOpen (pbSemName, cbSemName, options, pShRet): ercType

where

pbSemName
cbSemName

describe the semaphore name string. If *cbSemName* is 0, no name is assigned to the semaphore.

options

is a byte value describing the semaphore characteristics. The meaning of each bit is described below:

Bit Number	**Value**	**Meaning**
0	0	Any mutual exclusion semaphore.
	1	Critical section semaphore.
1	0	Any process locking the mutual exclusion semaphore can be terminated before it releases the lock.
	1	No process locking the mutual exclusion semaphore can be terminated until it releases the lock.

Bit Number	Value	Meaning
2	0	Any process obtaining the handle for the semaphore may change the semaphore state.
	1	Only processes within the context of the same user number that created the semaphore may change the semaphore state.

pShRet

is the memory address to which the 32-bit semaphore handle is returned.

Request Block

sShRet is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	406
12	options	2	
14	reserved	2	0
16	pbSemName	4	
20	cbSemName	2	
22	pShRet	4	
26	sShRet	2	4

SemQuery (sh, pbSemRecordRet, cbSemRecordRet, pbSemNameRet, cbSemNameRet): ercType

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases.*

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

SemQuery returns the semaphore record for the system semaphore specified by *sh*.

For details on semaphores, see the section entitled "Semaphores" in the *CTOS Operating System Concepts Manual.*

The Debugger uses this operation to return semaphore information when the end user types the **Code-Y** command. (See the *CTOS Debugger User's Guide* for details.)

Procedural Interface

SemQuery (sh, pbSemRecordRet, cbSemRecordRet, pbSemNameRet, cbSemNameRet): ercType

where

sh

is the 32-bit semaphore handle.

pbSemRecordRet
cbSemRecordRet

describe an area in memory where the system semaphore record is to be returned. A system semaphore record has the following format:

Offset	**Field**	**Size (Bytes)**	**Description**
0	lock	2	Indicates whether the semaphore is set or locked. If the low bit is 0, the semaphore is clear; if the low bit is 1, the semaphore is set or locked.
2	ownerTss	2	Specifies the TSS of the process that locked the mutual exclusion semaphore. The process is referred to as the *semaphore owner*.
4	userNum	2	The user number that created the semaphore.
6	useCount	2	For noncritical semaphores, the number of times the semaphore owner has locked the semaphore. Each time the semaphore owner clears the lock, the use count is decremented. The semaphore is not actually cleared, however, until the use count is 0.
8	openCount	2	The total number of times the semaphore has been opened. A user may open it more than once..

Offset	Field	Size (Bytes)	Description
10	priority	2	The original priority of the semaphore owner. If a process of a higher priority wants to lock the semaphore, the operating system raises the priority of the semaphore owner to be the same as the priority of the process wanting the semaphore. This allows the semaphore owner to complete its work so the semaphore can be more quickly available for the next process. When a semaphore owner clears the lock, its original priority is restored.
12	options	1	The description of the semaphore, as specified by the options at the time the semaphore was opened. (See the *options* parameter to SemOpen.)
13	flags	1	Reserved for future use.
14	pFirstWaiting	4	The memory address of the first waiting record. (For the format of a waiting record, see the SemEnumerateWaiting operation).
18	pLastWaiting	4	The memory address of the last waiting record.

Offset	Field	Size (Bytes)	Description
22	pSbName	4	The memory address of the semaphore name string (if one exists). The first byte contains the length of the name string.
26	openCountOwner	2	The number of times the owner opened the semaphore.
28	pNextUser	4	The memory address of the next semaphore record when another user has opened the semaphore. Each time a semaphore is opened by a new user, a semaphore record is allocated and queued in a list. This procedure tracks the number of times each user opens a semaphore. When a user terminates, for example, the semaphore termination procedure checks to see if the user has opened any semaphores. If the user was the first to open the semaphore, *openCount* is decremented by *openCountOwner*. If the user opened a semaphore already open, the semaphore record for the terminating user is located and removed from the queue. Then *openCount* is decremented by the number of times the terminating user opened the semaphore.

Offset	Field	Size (Bytes)	Description
28	pLastUser	4	The memory address of the last user number to lock the semaphore.

pbSemNameRet
cbSemNameRet

describe an area in memory where the semaphore name string is returned. The first byte contains the length of the name string.

Request Block

SemQuery is a system-common procedure.

This page intentionally left blank.

SemQueryProcessLock (pid, options, pShRet): ercType

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases.*

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

SemQueryProcessLock checks the list of system semaphore records to see if the specified process owns any locked semaphores. It returns the semaphore handle of the first locked semaphore it finds with characteristics matching the specified *options*.

If the process does not own any locked semaphores with the specified options, status code 13953 ("Invalid semaphore handle") is returned.

The termination process calls SemQueryProcessLock to determine whether a terminating process owns any nonterminatable, critical section semaphores. If it finds a locked semaphore, it attempts to obtain the lock, causing it to wait in the run queue. As soon as it obtains the lock, it releases it and calls SemQueryProcessLock again to check if the process owns any other locks.

Because SemQueryProcessLock is a subroutine of the termination process, it runs at a higher priority than the process being terminated.

Procedural Interface

SemQueryProcessLock (pid, options, pShRet): ercType

where

pid

is the identifier of the process being queried.

options

is a byte value describing the characteristics of the locked semaphore. Each bit indicates the following:

Bit Number	**Value**	**Meaning**
0	0	Mutual exclusion semaphore
	1	Critical section semaphore
1	0	Terminatable semaphore
	1	Nonterminatable semaphore

pShRet

is the memory address to which the 32-bit semaphore handle is returned.

Request Block

SemQueryProcessLock is a system-common procedure.

SemSet (sh): ercType

NOTES:

1. *This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.*
2. *The SemMuxWait, SemNotify, SemSet, and SemWait operations should be used for signaling, not for mutual exclusion.*

Description

SemSet sets the signaling semaphore specified by *sh* regardless of the present state of the semaphore.

For details on semaphores, see the section entitled "Semaphores" in the *CTOS Operating System Concepts Manual.*

Procedural Interface

SemSet (sh): ercType

where

sh

is the 32-bit semaphore handle.

Request Block

SemSet is a system-common procedure.

This page intentionally left blank.

SemWait (sh, timeout): ercType

NOTES:

1. *This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.*
2. *The SemMuxWait, SemNotify, SemSet, and SemWait operations should be used for signaling, not for mutual exclusion.*

Description

SemWait causes the calling process to wait for the signaling semaphore specified by *sh* to be cleared. Then it allows the caller to continue execution. If the semaphore is already clear, no waiting is necessary. Otherwise the caller waits at its default response exchange until either the semaphore is cleared with the SemClear operation by the process responsible for setting it or a timeout occurs. (See SemClear.)

SemWait differs from SemMuxWait in that the latter allows the calling process to wait for more than one signaling semaphore. (See SemMuxWait.)

For details on semaphores, see the section entitled "Semaphores" in the *CTOS Operating System Concepts Manual.*

Procedural Interface

SemWait (sh, timeout): ercType

where

sh

is the 32-bit semaphore handle.

timeout

specifies the amount of time that the process will wait for the semaphore to be cleared. The timeout values and their meanings are

Value	**Meaning**
0	Do not wait. If the semaphore is not clear, return status code 13955 ("Semaphore timeout").
-1	Wait indefinitely for the semaphore to be cleared.
Greater than 0	Wait for the specified amount of time in units of 100 milliseconds.

Request Block

SemWait is a system-common procedure.

Send (exchange, pMsg): ercType

Description

The Send primitive checks whether processes are queued at the specified exchange. If processes are queued, then the process that is queued first is removed from the queue, given the memory address of the message, and placed into the ready state. If such a process has a higher priority than the calling process, it is scheduled for immediate execution and the calling process remains preempted until the higher priority process reenters the waiting state.

If no processes are waiting at the exchange, then the message is queued at the exchange.

Send should be used within the context of a single user number. (For filter processes, see ForwardRequest.)

Procedural Interface

Send (exchange, pMsg): ercType

where

exchange

is the identification of the exchange to which the message is sent.

pMsg

is the memory address of the message (or a 4-byte field of information whose interpretation is agreed upon by the sending and receiving process).

Request Block

Send is a Kernel primitive.

This page intentionally left blank

SendBreakC (pBswa): ercType

Description

SendBreakC sends a break signal on the communications line previously opened under the Sequential Access Method.

SendBreakC delays 1/5 of a second (or longer if there are other characters remaining to be transmitted) before returning to the calling operation. There is no way to avoid the 1/5 second delay, although longer periods of blocking may be prevented by using CheckpointBsAsyncC before calling SendBreakC.

(For more information on the communications programming interfaces, see "Communications Programming" in the *CTOS Operating System Concepts Manual.*)

Procedural Interface

SendBreakC (pBswa): ercType

where

pBswa

is the memory address of the Byte Stream Work Area (BSWA).

Request Block

SendBreakC is an object module procedure.

This page intentionally left blank

SeqAccessCheckpoint (seqHandle, pCbResidual): ercType

Description

SeqAccessCheckpoint causes the caller to wait until the all of the data written by previous SeqAccessWrite calls has been successfully transferred to the medium, or until an exception condition occurs that prevents the transfer of all buffered data. (See SeqAccessWrite.)

This operation is performed by the Sequential Access Service. (See "Sequential Access Service" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

SeqAccessCheckpoint (seqHandle, pCbResidual): ercType

where

seqHandle

is a sequential access device handle returned by a previous call to the SeqAccessOpen operation. (See SeqAccessOpen.)

pCbResidual

is the memory address of a double word to which the count of bytes NOT successfully transferred to the medium is returned. This count may reflect the number of bytes in the buffers of either the Sequential Access Service or the sequential access device. This count is greater than zero only if an exception condition occurs that prevents the successful transfer of all buffered data to the medium.

Request Block

sCbResidual is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33234
12	seqHandle	2	
14	reserved	10	
24	pCbResidual	4	
28	sCbResidual	2	4

SeqAccessClose (seqHandle): ercType

Caution: *Before calling SeqAccessClose, an application should always prompt the user to remove the medium. Otherwise, the medium could be overwritten by another user.*

Description

SeqAccessClose releases a sequential access device from the exclusive access rights granted by a previous call to SeqAccessOpen. (See SeqAccessOpen.) The position of any medium in the device is not affected unless there is buffered data that must still be transferred to the medium. If buffered data transfer operations are in progress, SeqAccessClose waits for their completion before it closes the device.

This operation is performed by the Sequential Access Service. (See "Sequential Access Service" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

SeqAccessClose (seqHandle): ercType

where

seqHandle

is a sequential access device handle returned by a previous call to the SeqAccessOpen operation. (See SeqAccessOpen.)

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33227
12	seqHandle	2	

SeqAccessControl (seqHandle, ctrlFunction, ctrlQualifier, pCbResidual): ercType

Description

SeqAccessControl causes the Sequential Access Service to perform medium positioning operations, as well as other operations that do not involve the transfer of user data to the medium. For example, SeqAccessControl may cause the sequential access device to rewind, unload, erase, or retension the medium.

If buffered data transfer operations are in progress, SeqAccessControl awaits their completion before attempting the control operation requested.

Status code 9017 ("Device is write protected") is returned when an Erase Medium, Write Filemark, or Erase Gap operation is attempted on a write-protected medium. Status code 9037 ("Device is read-only") is returned when an Erase Medium, Write Filemark, or Erase Gap operation is attempted with a device that was opened in read mode.

Some sequential access devices cannot perform all of the provided control operations. If the requested operation is neither supported nor required by the device (for example, retension on a DDS device), no operation takes place and status code zero ("Erc OK") is returned. Otherwise, status code 9074 ("Illegal command") is returned if the device cannot perform the requested operation.

This request is performed by the Sequential Access Service. (See "Sequential Access Service" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

NOTE: Medium erasure may take a very long period of time for some devices. For example, two hours are required to erase a 60 meter DDS cartridge.

Procedural Interface

SeqAccessControl (seqHandle, ctrlFunction, ctrlQualifier, pCbResidual): ercType

where

seqHandle

is a sequential access device handle returned by a previous call to the SeqAccessOpen operation (see SeqAccessOpen.)

ctrlFunction

is a value that specifies the control operation to be performed. Each value indicates the following:

Command	**Value**	**Description**
Rewind	1	Rewinds the medium to its load point (the beginning of the usable portion of the medium). If the *ctrlQualifier* parameter is 0, this operation does not finish until the medium is positioned at its load point. If the *ctrlQualifier* parameter is −1, SeqAccessControl returns control to the caller as soon as the sequential access device has accepted the command; this leaves the caller free to perform other tasks while the medium is rewinding. The caller may use the SeqAccessStatus request to determine when the medium positioning operation is complete. (See SeqAccessStatus.)

Command	Value	Description
Unload	2	Rewinds the medium to its load point. With some devices, this command also causes physical ejection of the medium. If the *ctrlQualifier* parameter is 0, this operation does not complete until the medium is returned to its load point. If the *ctrlQualifier* parameter is −1, this operation returns control to the caller as soon as the sequential access device has accepted the command; this leaves the caller free to perform other tasks while the medium is being unloaded. The SeqAccessStatus operation may be used to determine when the medium is unloaded. (See SeqAccessStatus.)
Retension	3	Moves the medium forward from its current position to its endpoint and then rewinds the medium to its load point. If the *ctrlQualifier* parameter is zero, this operation does not complete until the medium is positioned at its load point. If the *ctrlQualifier* parameter is −1, the operation completes as soon as the sequential access device accepts the command; this leaves the user program free to perform other tasks while the medium is being retensioned. The SeqAccessStatus operation may be used to determine when the retensioning is complete. (See SeqAccessStatus.)

Command	Value	Description
Erase Medium	4	Erases the remainder of the medium from the current position to the physical end of the medium. If the *ctrlQualifier* parameter is zero, this operation does not complete until the sequential access device indicates that the medium is at its load point. If the *ctrlQualifier* parameter is −1, the operation completes as soon as the sequential access device has accepted the command; this leaves the user program free to perform other tasks while the medium is being erased. The SeqAccessStatus operation may be used to determine when the medium erasure is complete.
Write Filemark	5	Causes a specific number of consecutive filemarks to be written to the medium. The *ctrlQualifier* parameter specifies the number of filemarks to be written. Note that *ctrlQualifier* must be positive; filemarks may not be written when the medium is moving backwards.

Scan Filemarks	6	Searches for a specific number of consecutive filemarks. The *ctrlQualifier* parameter specifies the number of filemarks to be located. If *ctrlQualifier* is positive, the medium is moved forwards (in the direction of the logical end of medium). When the desired number of consecutive filemarks are found, the medium is left positioned after the last filemark of the set. If the ctrlQualifier parameter is negative, movement is backwards (in the direction of the logical beginning of the medium). In this case, when the desired number of consecutive filemarks are found, the medium is left positioned before the first filemark of the set. The Scan Filemarks operation is terminated if an end of data, beginning of medium, end of medium, or other exception condition is encountered before the desired number of filemarks have been located.
Space Records	7	Skips over a specific number of records on the medium. The *ctrlQualifier* parameter specifies the number of records to be skipped. If *ctrlQualifier* is positive, the medium is moved forwards (in the direction of the logical end of the medium). When the Space Records operation is complete, the medium is left positioned after the last record specified. If *ctrlQualifier* is negative, the medium is moved backwards (in the direction of the logical beginning of medium). In this case, when the Space Records operation is complete, the medium is left positioned before the last record specified. This operation is terminated if a filemark, end of data, beginning of medium, end of medium, or other exception condition is encountered before the desired number of records have been skipped.

Command	Value	Description
Erase Gap	8	Creates a specific number of gaps forward from the current position of the medium. The *ctrlQualifier* parameter specifies the number of gaps to be created. The size of the created gap is hardware dependent and may vary with the medium positioning speed of the device. SeqAccessModeQuery may be used to determine the size of the erase gap. (See SeqAccessModeQuery.) Note that the *ctrlQualifier* parameter must be a positive number; the medium may not be erased in a backwards direction. Sequential access devices that do not support individual erase gaps (for example, QIC devices) return status code zero and perform no operation.

ctrlQualifier

is an optional additional parameter (signed word) that modifies the action of the specified control function. If a control function does not use this argument, it is ignored.

pCbResidual

is the memory address of a double word to which the count of bytes remaining in both the Sequential Access Service buffers and the sequential access device buffers is returned.

Request Block

sCbResidual is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33230
12	seqHandle	2	
14	ctrlFunction	2	
16	ctrlQualifier	2	
18	reserved	6	
24	pCbResidual	4	
28	sCbResidual	2	4

This page intentionally left blank

SeqAccessDiscardBufferData

SeqAccessDiscardBufferData (seqHandle): ercType

Description

SeqAccessDiscardBufferData discards data from the buffers of the Sequential Access Service. The sequential access device must have been previously opened by the SeqAccessOpen operation. (See SeqAccessOpen.)

Using SeqAccessDiscardBufferData, an application can remove data that is left in the Sequential Access Service buffers when an early–warning end-of-medium or other exception condition occurs. This permits the application to logically terminate the data on the medium with a filemark written by the SeqAccessControl operation. If the buffers are not emptied of data by either a SeqAccessDiscardBufferData or SeqAccessRecoverBufferData operation, SeqAccessControl attempts to transfer all buffered data to the medium before the filemark is written.

SeqAccessDiscardBufferData and SeqAccessRecoverBufferData are similar in that they both remove data from the buffers of the Sequential Access Service. Unlike the latter operation, however, SeqAccessDiscardBufferData does not recover buffer data.

If any data remains in the device buffers when a SeqAccessDiscardBufferData operation is performed, status code 9061 ("Device buffer not empty") is returned.

This operation is performed by the Sequential Access Service. (See "Sequential Access Service" in the *CTOS Programming Guide.*) The service must be installed for this operation to be used.

Procedural Interface

SeqAccessDiscardBufferData (seqHandle): ercType

where

seqHandle

is a sequential access device handle returned by a previous SeqAccessOpen operation.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33236
12	seqHandle	2	

SeqAccessModeQuery (seqHandle, pModeParametersRet, sModeParametersMax): ercType

Description

SeqAccessModeQuery returns information about the current operating characteristics of the sequential access device. For example, it may report the recording density of the medium, whether the device is write-protected, or whether the device is operating in buffered mode. The application must provide the handle of the sequential access device for which information is to be returned.

Using SeqAccessModeQuery, an application can determine whether the operating characteristics originally specified by a SeqAccessOpen or SeqAccessModeSet operation have been accepted. (See SeqAccessOpen and SeqAccessModeSet.)

If an application specifies default operating characteristics in a call to SeqAccessOpen or SeqAccessModeSet, it can also use SeqAccessModeQuery to examine the default settings. If the application needs to change these settings, it can subsequently call SeqAccessModeSet.

This operation is performed by the Sequential Access Service. (See "Sequential Access Service" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

SeqAccessModeQuery (seqHandle, pModeParametersRet,
sModeParametersMax): ercType

where

seqHandle

is a sequential access device handle returned by a previous call to SeqAccessOpen.

pModeParametersRet
sModeParametersMax

describe the memory location where a parameter block is returned. This block specifies values for the current operating characteristics of the sequential access device. For the format of this parameter block, see the *pModeParameters* parameter of SeqAccessOpen.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33228
12	seqHandle	2	
14	reserved	4	
18	pModeParametersRet	4	
22	sModeParametersMax	2	

This page intentionally left blank

SeqAccessModeSet (seqHandle, pModeParameters, sModeParameters): ercType

Description

SeqAccessModeSet configures operating characteristics of the sequential access device. Such characteristics include buffered or non-buffered modes, medium movement speeds, and the medium recording density. The information block used to communicate this information to the Sequential Access Service is also used for the return of information by the SeqAccessModeQuery operation. Consequently, some of the fields in this block may only be read; they cannot be changed by SeqAccessModeSet. If the caller tries to change a read-only value, SeqAccessModeSet leaves this value unchanged, and does not return an error message.

The application specifies operating characteristics of the device by supplying a parameter block at *pModeParameters*. By calling SeqAccessModeQuery, an application can check whether each supplied value has been accepted. (See SeqAccessModeSet.)

Status code 9022 ("Invalid mode parameters") is returned if any mode parameter at *pModeParameters* is invalid.

This operation is performed by the Sequential Access Service. (See "Sequential Access Service" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

SeqAccessModeSet (seqHandle, pModeParameters, sModeParameters): ercType

where

seqHandle

is a sequential access device handle returned by a previous call to the SeqAccessOpen operation (see SeqAccessOpen.)

pModeParameters
sModeParameters

describe the memory location of a parameter block. This block specifies values for the desired current operating characteristics of the sequential access device. For the format of this parameter block, see the *pModeParameters* parameter of SeqAccessOpen.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33229
12	seqHandle	2	
14	reserved	4	
18	pModeParameters	4	
22	sModeParameters	2	

This page intentionally left blank

SeqAccessOpen (pSeqHandleRet, pbDeviceName, cbDeviceName, pbPassword, cbPassword, accessMode, pModeParameters, sModeParameters): ercType

Description

SeqAccessOpen provides exclusive access to the specified sequential access device and returns a sequential access handle to be used in subsequent operations. The name of the requested device in conjunction with device name routing determines which Sequential Access Service is to receive this request (for example, the service installed at the local workstation, or one of possibly several services installed at the server).

The application can specify the operating characteristics of the device by supplying a parameter block at *pModeParameters*. If the application specifies an invalid value in a parameter block field, SeqAccessOpen in some cases rejects the parameter and returns status code 9022 ("Invalid mode parameter"). In other cases, SeqAccessOpen substitutes the default value for the one specified. By subsequently calling SeqAccessModeQuery, an application can check whether each supplied value has been accepted.

If the application does not supply a parameter block, SeqAccessOpen uses the default setting for each operating characteristic. To examine these default settings, the application can call SeqAccessModeQuery. (See SeqAccessModeQuery.) If a mode parameter block is not specified or if the service buffer size is not specified, SeqAccessOpen sets the service buffer size to the greater of a hard-coded default or the block size (or, if variable length records are to be used, the maximum record size).

Status code 9035 ("Invalid device or file specification") is returned if the specified device name is invalid. Status code 9036 ("Device in use") is returned if the device is already in use. Status code 9040 ("Invalid mode") is returned if the application specifies a mode other than "mr" or "mm".

This operation is performed by the Sequential Access Service. (See "Sequential Access Service" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

SeqAccessOpen (pSeqHandleRet, pbDeviceName, cbDeviceName, pbPassword, cbPassword, accessMode, pModeParameters, sModeParameters): ercType

where

pSeqHandleRet

is the memory address of a word to which the sequential access device handle is returned. Any subsequent operations that access this device must use this handle.

pbDeviceName
cbDeviceName

describe the name of the requested sequential access device. Device names are chosen by the user at the time the Sequential Access Service is installed. Typical device names include the following:

Name	**Meaning**
[QIC]	Quarter-inch cartridge (may be either QIC-02 or SCSI interfaces).
[TAPE]	Half-Inch reel-to-reel (may be either SCSI, DP or SP interfaces).
[DDS]	4 mm digital data storage (also called R-DAT).
[HITC]	Half-inch cartridge.

pbPassword
cbPassword

describe the password that authorizes exclusive access to the specified sequential access device.

accessMode

is a code which, when set to "mr", indicates that permission is granted only to read data from the sequential access device. When set to "mm", this code indicates that permission is granted to read and write data from/to the sequential access device.

pModeParameters
sModeParameters

describe a parameter block used to configure operating modes of the sequential access device. Note that this parameter block is also used by the SeqAccessModeQuery and SeqAccessModeSet operations. Some of its values cannot be changed by the caller. This parameter block has the following format:

Offset	Field	Size (Bytes)	Description
0	fWriteProtected	1	A read-only value. TRUE means data may not be written to the sequential access device.
1	fVariableLength	1	TRUE indicates that the device is operating in variable-length mode. FALSE indicates that the device is operating in fixed-length mode.

Offset	Field	Size (Bytes)	Description
2	fUnbuffered	1	TRUE requests that the Sequential Access Service allocate no buffer space. When *fUnbuffered* is TRUE, data is transferred directly between the device and the buffers supplied in calls to SeqAccessRead and SeqAccessWrite. FALSE (the default setting) selects buffered mode. In this mode, the Sequential Access Service allocates space for buffers. Data then passes through these buffers when it is transferred between the device and the user program.
3	fSuppressDefaultModeOnOpen	1	A flag. If TRUE, current operating characteristics of the device are NOT reset to the default characteristics before the parameters supplied by SeqAccessOpen are applied.
4	mediumSpeed	2	A code that sets the transport speed for the medium. Zero is the default value; the speed increases as this value becomes larger. For QIC and DDS devices, this field has no effect. For devices that use SRP/XE DP or SP half-inch tape, a value of 0 indicates low speed, while a value of 1 indicates high speed.

Offset	Field	Size (Bytes)	Description
6	mediumDensity	2	A code that sets the medium recording density for write operations. For QIC and DDS devices, this field has no effect. For devices that use 9 track half-inch tape, each value in this field indicates the following recording density:

Value	Density
1	800 bpi NRZI
2	1600 bpi PE
3	6250 bpi GCR
6	3200 bpi PE

Offset	Field	Size (Bytes)	Description
8	totalBlocks	4	A read-only value. This code is an estimate of the number of records that can be written to the entire length of the medium. If no estimate is available (as is the case with fixed records), 0 is returned.
12	blockSize	4	The size of the physical record to be written to the medium. This value is not applicable for some devices, nor is it applicable in variable-length mode. In fixed-length mode, this value must also be a multiple of *minRecordSize*.

Offset	Field	Size (Bytes)	Description
16	minRecordSize	2	The smallest variable-length record that may be written to the medium. Or, the size of all fixed-length records that will be written to the medium.
18	maxRecordSize	4	For variable-length records, this value is the largest record that may be written to the medium. For fixed-length records, this value must be equal to *minRecordSize*.
22	deviceBufferSize	4	If available, the size (in bytes) of the buffers present in the device itself.
26	serviceBufPoolSize	4	The amount of buffer space allocated in the Sequential Access Service.
30	serviceBuffers	2	The number of separate buffers into which the buffer space of the Sequential Access Service is divided.
32	serviceBufferSize	2	The size of each buffer created by the division of the Sequential Access Service buffer space.
34	writeBufThreshold	4	Threshold amount of buffer data for write operations. When the service buffer contains this amount or greater, data is transferred to the device until the service buffers are empty, or until an exception condition is encountered.
38	readBufThreshold	4	Threshold amount of buffer data for read operations. When the amount of data in the service buffer is less than this amount, the service reads data from the device until its buffers are full, or until an exception condition is encountered.

Offset	Field	Size (Bytes)	Description
42	fBufDataUnrecoverable	1	A read-only flag. TRUE indicates that SeqAccessRecoverBufferData cannot recover data from the buffers of the device.
43	fDisableAutoSpeed	1	A flag. TRUE inhibits the ability of the device to optimize the medium transport speed. FALSE allows the device to optimize the medium transport speed.
44	fBufRecoverLIFO	1	A flag. TRUE specifies that buffer data is recovered in a last-in, first-out order. FALSE specifies that device buffer data is recovered in a first-in, first-out order.
45	fCheckpointEOM	1	A flag. If TRUE, the device attempts to empty its buffers upon reaching the early-warning end-of-medium. If FALSE, the device does not attempt to transfer its buffer data to the medium when the early-warning end-of-medium is encountered.
46	fDataCompression	1	A flag. If TRUE, data compression is enabled.
47	gapSize	1	A value that specifies the size of the short erase gap created when the SeqAccessControl request performs an Erase Gap function.
48	bufferSizeEOM	4	A read-only value. The number of bytes by which the device reduces its internal buffer capacity when it encounters an early-warning end-of-medium condition.

Offset	Field	Size (Bytes)	Description
52	fReportSoftErrors	1	A flag. If TRUE, status information about recovered errors is requested.
53	fDisableErrorCorrection	1	A flag. If TRUE, no error correction computation is applied to recover read data. Instead, a read operation is retried if *fDisableReadRetries* is FALSE (see below).
54	fDisableReadRetries	1	A flag. If TRUE, neither the Sequential Access Service nor the device attempts to retry a read operation if an error occurs.
55	fDisableWriteRetries	1	A flag. If TRUE, neither the Sequential Access Service nor the device attempts to retry a write operation if an error occurs.
56	readRetryLimit	2	The number of times an unsuccessful read operation is to be retried. If a zero value is supplied and *fDisableReadRetries* is FALSE, the default retry limit for the device is used.
58	writeRetryLimit	2	The number of times an unsuccessful write operation is to be retried. If a zero value is supplied and *fDisableWriteRetries* is FALSE, the default retry limit for the device is used.

Request Block

sSeqHandleRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	3
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33226
12	accessMode	2	
14	reserved	4	
18	pbDeviceName	4	
22	cbDeviceName	2	
24	pbPassword	4	
28	cbPassword	2	
30	pModeParameters	4	
34	sModeParameters	2	
36	pSeqHandleRet	4	
40	sSeqHandleRet	2	2

This page intentionally left blank

SeqAccessRead (seqHandle, pDataRet, sDataMax, pCbResidual): ercType

Note: A program can use SeqAccessRead to read raw data on tapes that do not necessarily follow the Unisys archive format.

Description

SeqAccessRead reads data from the sequential access device and places it in a caller-specified buffer. The device must have been previously opened by the SeqAccessOpen operation. (See SeqAccessOpen.)

SeqAccessRead and SeqAccessWrite operations will work with devices opened in modify mode. However, the caller must ensure that the device and Sequential Access Service buffers are empty before it reverses the direction of the data transfer. These buffers are always empty after the successful completion of any SeqAccessControl operation. After a successful SeqAccessCheckpoint operation, these buffers are always empty, provided a write operation is followed by a read operation.

Status code 9007 ("Medium blank") is returned if the medium in the sequential access device does not contain recorded data. If a SeqAccessRead follows a SeqAccessWrite (or vice-versa) and the buffers are not empty, status code 9084 ("Residual data") is returned.

This operation is performed by the Sequential Access Service. (See "Sequential Access Service" in the *CTOS Programming Guide.*) The service must be installed for this operation to be used.

Procedural Interface

SeqAccessRead (seqHandle, pDataRet, sDataMax, pCbResidual): ercType

where

seqHandle

is a sequential access device handle returned by a previous SeqAccessOpen operation.

pDataRet
sDataMax

describe a buffer to which data read from the sequential access device is returned. If the sequential access device operates in fixed length mode, the size of the buffer must be a multiple of the record size. (In fixed-length mode, record sizes are not variable.)

pCbResidual

is the memory address of a double word to which the difference between the requested transfer length and the actual transfer length is returned. In fixed-length mode, a nonzero value is returned only when a filemark, end of data, end of medium or other exception condition is encountered before the requested amount of data has been transferred. In these cases, the SeqAccessRead operation also returns a nonzero status code.

In variable length mode, a nonzero *cbResidual* value may be used to calculate the size of the variable-length record just read. If *cbResidual* is less than zero, the physical record on the medium was larger than the data area allocated by the caller. The record can be read in its entirety by performing a SeqAccessControl operation that specifies a Space Record operation with a *ctrlQualifier* of -1, followed by a SeqAccessRead operation with a data area large enough to accommodate the record. If *cbResidual* is greater than or equal to zero, the data area was large enough to accommodate the record, and *cbResidual* reflects the unused portion of the data area. Unless the caller encounters an exception condition, a zero status code and a *cbResidual* value greater than or equal to zero are normal conditions in variable-length mode.

Request Block

sCbResidual is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33233
12	seqHandle	2	
14	reserved	4	
18	pDataRet	4	
22	sDataMax	2	
24	pCbResidual	4	
28	sCbResidual	2	4

This page intentionally left blank.

SeqAccessRecoverBufferData (seqHandle, pDataRet, sDataMax, pCbResidual): ercType

Description

SeqAccessRecoverBufferData transfers data from the buffers of the Sequential Access Service to a caller-specified buffer. The sequential access device must have been previously opened by the SeqAccessOpen operation. (See SeqAccessOpen.)

SeqAccessRecoverBufferData allows an application to recover data that is left in the device buffers and the Sequential Access Service buffers when an early-warning end-of-medium or other exception condition occurs. This permits the application to logically terminate the data on the medium with a filemark written by the SeqAccessControl operation. If the buffers have not been emptied of data either by this operation or by SeqAccessDiscardBufferData, SeqAccessControl attempts to transfer all buffered data to the medium before the filemark is written. Once a filemark has been written and a new medium has been mounted, the application can resume writing the data that was recovered.

SeqAccessRecoverBufferData and SeqAccessDiscardBufferData are similar in one respect: they both remove data from the Sequential Access Service buffers. However, SeqAccessRecoverBufferData returns the data to the caller.

This operation is performed by the Sequential Access Service. (See "Sequential Access Service" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

SeqAccessRecoverBufferData (seqHandle, pDataRet, sDataMax, pCbResidual): ercType

where

seqHandle

is a sequential access device handle returned by a previous call to the SeqAccessOpen operation. (See SeqAccessOpen.)

pDataRet
sDataMax

describe a memory area to which Sequential Access Service and device buffer data is returned.

pCbResidual

is the memory address of a double word to which the difference between the requested transfer length and the actual transfer length is returned.

Request Block

sCbResidual is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33235
12	seqHandle	2	
14	reserved	4	
18	pDataRet	4	
22	sDataRet	2	
24	pCbResidual	4	
28	sCbResidual	2	4

This page intentionally left blank

SeqAccessStatus (seqHandle, pStatusRet, sStatusMax, pCbResidual): ercType

Description

SeqAccessStatus allows the caller to determine the current status of the sequential access device and its medium. The device must have been previously opened by the SeqAccessOpen operation. (See SeqAccessOpen.)

This operation is performed by the Sequential Access Service. (See "Sequential Access Service" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

SeqAccessStatus (seqHandle, pStatusRet, sStatusMax, pCbResidual): ercType

where

seqHandle

is a sequential access device handle returned by a previous SeqAccessOpen operation.

pStatusRet
sStatusMax

describe the memory area of a double word to which status information is returned. In the first word, each bit, if set, indicates the following:

Bit Mask	**Field**	**Description**
01h	mediumLoaded	The medium is present in the device.
02h	writeProtect	No data transfers to the medium may be performed.
04h	busy	The device is busy performing a previously requested operation.
08h	ready	The device is ready for data transfer.
10h	endOfMedium	The medium is at the end-of-medium position.
20h	beginningOfMedium	The medium is at the beginning-of-medium position.
40h	onLine	The presence of the device is detected and the device is available. The medium may or may not be ready.
80h	reset	The device has been turned on, or unit attention (for example, medium insertion or removal) has occurred.

The second word holds the count of recovered errors. This count is not available for all devices. It is reset to zero when the direction of data transfer changes, or when the medium is rewound, removed or inserted.

pCbResidual

is the memory address of a double word to which the count of bytes in the Sequential Access Service and device buffers is returned.

Request Block

sCbResidual is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33231
12	seqHandle	2	
14	reserved	4	
18	pStatusRet	4	
22	sStatusMax	2	
24	pCbResidual	4	
28	sCbResidual	2	4

This page intentionally left blank

SeqAccessVersion (pbDeviceName, cbDeviceName, pVersionInfoRet, sVersionInfoMax): ercType

Description

SeqAccessVersion returns version information about the Sequential Access Service that controls the named device. It also provides information about other devices under the control of this Sequential Access Service.

This operation is performed by the Sequential Access Service. (See "Sequential Access Service" in the *CTOS Programming Guide.*) The service must be installed for this operation to be used.

Procedural Interface

SeqAccessVersion (pbDeviceName, cbDeviceName, pVersionInfoRet, sVersionInfoMax): ercType

where

pbDeviceName
cbDeviceName

describe the name of the requested sequential access device.

pVersionInfoRet
sVersionInfoMax

describe an area to which version information about the Sequential Access Service is returned. This information has the following format:

Offset	**Field**	**Size (Bytes)**	**Description**
0	majorVersion	2	The major version level.
2	minorRevision	2	The minor revision level.
4	subRevision	2	The subrevision level (for example, the value 3 in the expression "version 12.1.3").
6	devices	2	The count of devices served by the Sequential Access Service responsible for the device specified in the SeqAccessVersion call.
8	versionText	65	A text string which contains version information. The first byte is the string length.
73	node	13	Name of the node at which the Sequential Access Service is installed. The first byte is the string length.
86	processor	13	Name of the processor at which the Sequential Access Service is installed. This field is set to zero if the Sequential Access Service is installed on a single-processor system.

Offset	Field	Size (Bytes)	Description
99	fServer	1	A flag that is TRUE if the Sequential Access Service is installed on a server. This can be used by the application to determine if it is more appropriate to consult the device configuration file on the local or server system volume.
100	deviceNames	varies	An array of 13-byte entries that describe the names of all the devices under the control of the Sequential Access Service. The count of devices determines the number of entries in this array. The first byte of each entry is the length of the entry.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33225
12	reserved	6	
18	pbDeviceName	4	
22	cbDeviceName	2	
24	pVersionInfoRet	4	
28	sVersionInfoMax	2	

SeqAccessWrite (seqHandle, pData, sData, pCbResidual): ercType

Caution: *A program can use SeqAccessWrite to write to tapes that do not necessarily follow the Unisys format.*

Description

SeqAccessWrite writes data to the medium mounted on a sequential access device. The device must have been previously opened by the SeqAccessOpen operation. (See SeqAccessOpen.)

The caller specifies the size and location of the data to be written, and SeqAccessWrite returns the amount of data unsuccessfully transferred.

SeqAccessWrite and SeqAccessRead operations will work with devices opened in modify mode. However, the caller must ensure that the device and Sequential Access Service buffers are empty before it reverses the direction of the data transfer. These buffers are always empty after the successful completion of any SeqAccessControl operation. After a successful SeqAccessCheckpoint operation, these buffers are always empty, provided a write operation is followed by a read operation.

Status code 9037 ("Device is read-only") is returned if the device was previously opened in read mode instead of modify mode. If a SeqAccessWrite follows a SeqAccessRead (or vice-versa) and the buffers are not empty, status code 9084 ("Residual data") is returned.

This operation is performed by the Sequential Access Service. (See "Sequential Access Service" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

SeqAccessWrite (seqHandle, pData, sData, pCbResidual): ercType

where

seqHandle

is a sequential access device handle returned by a previous SeqAccessOpen operation.

pData
sData

describe a memory area containing the data to be written to the sequential access device. If the sequential access device is configured to operate in fixed-length mode, the amount of data to be written must be a multiple of the record size. If the device is operating in variable-length mode, a physical record (block) is written whose length is equal to *sData*.

pCbResidual

is the memory address of a double word to which the amount of data unsuccessfully transferred to the medium is returned. If the sequential access device is operating in buffered mode, the value returned may exceed the amount of data originally specified in the call to SeqAccessWrite.

A nonzero value that is greater than the size of the data supplied to SeqAccessWrite indicates that data from a previous SeqAccessWrite operation still resides in the Sequential Access Service or device buffers. A subsequent SeqAccessWrite, SeqAccessCheckpoint, or SeqAccessControl operation (with the Write Filemark command specified) will attempt to transfer this data to the medium.

Request Block

sCbResidual is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33232
12	seqHandle	2	
14	reserved	4	
18	pData	4	
22	sData	2	
24	pCbResidual	4	
28	sCbResidual	2	4

This page intentionally left blank

SerialNumberQuery (iProcessor, pIdRet): ercType

NOTE: This request is the same as the system library procedure SerialNumberOldOsQuery. For backwards compatibility, use SerialNumberOldOsQuery.

Description

SerialNumberQuery returns the boot ROM's unique 32-bit serial number at the specified address.

If the workstation does not have a serialized boot ROM, status code 7 ("Not implemented") is returned.

Procedural Interface

SerialNumberQuery (iProcessor, pIdRet): ercType

where

iProcessor

is 0 for workstations. On a shared resource processor, this value is the slot number (for example 71h) of the processor board the caller is querying. (See the *CTOS System Administration Guide* for details on slot numbering.)

pIdRet

is the memory address of a 4 byte area where the boot ROM's unique 32-bit serial number is returned.

Request Block

sIdRet is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	1
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	391
12	iProcessor	2	
14	reserved	4	
18	pIdRet	4	
22	sIdRet	2	4

SerialNumberOldOsQuery (iProcessor, pIdRet): ercType

NOTE: This system library procedure is the same as the request SerialNumberQuery but is provided for backwards compatibility.

Description

SerialNumberOldOsQuery returns the boot ROM's unique 32-bit serial number at the specified address. To obtain the serial number, this object module procedure calls the request SerialNumberQuery.

On workstations, if SerialNumberQuery returns status code 31 ("No such request code") or status code 33 ("Service not available"), SerialNumberOldOsQuery obtains the serial number directly from the boot ROM. The only exception is on a 386i workstation: if SerialNumberQuery is not a supported request and if the workstation has 16 megabytes or more of memory, this procedure returns status code 7 ("Not implemented") if called from an protected mode program. (If the program is linked as a real mode program then this procedure will work on any 386i workstation.) Status code 7 ("Not implemented") is also returned if a workstation does not have a serialized boot ROM.

On a shared resource processor, if status code 31 or status code 33 is returned, SerialNumberOldOsQuery does not try to obtain the serial number from the boot ROM. Instead, it returns status code 7.

Procedural Interface

SerialNumberOldOsQuery (iProcessor, pIdRet): ercType

where

iProcessor

is 0 for workstations. On a shared resource processor, this value is the slot number (for example 71h) of the processor board the caller is querying. If *iProcessor* is 0 on a shared resource processor, the serial number of the local processor is returned.

pIdRet

is the memory address of a 4 byte area where the boot ROM's unique 32-bit serial number is returned.

Request Block

SerialNumberOldOsQuery is an object module procedure.

ServeRq (requestCode, exchange): ercType

Caution: *When running on a protected mode operating system, this operation should be used only by programs executing in protected mode.*

Description

ServeRq is used by a system service that is dynamically installed to serve the specified request code. Future requests containing the specified request code are queued at the specified exchange. Status code 31 ("No such request code") is returned if the request is not defined. (For details on defining requests, see "System Services Management" in the *CTOS Operating System Concepts Manual*.)

Specifying exchange 0 indicates that the calling system service is no longer serving the specified request code. However, currently queued requests are not removed from the queue at the exchange that was formerly associated with the specified request code. Status code 33 ("Service not available") is returned to future requests containing the request code.

Procedural Interface

ServeRq (requestCode, exchange): ercType

where

requestCode

is the request code.

exchange

is the service exchange number or 0.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum		
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	99
12	requestCode	2	
14	exchange	2	

Set386TrapHandler (iTrap, pTrapHandler): ercType

Caution: *Set386TrapHandler is supported in protected mode only on operating systems running on the 80386 microprocessor.*

Description

Set386TrapHandler is used to set a 386 trap handler to serve software generated interrupts (traps) and exceptions.

Unlike SetIntHandler, Set386TrapHandler is always raw (not mediated). The trap handler is part of the process context for all processes in a partition as opposed to being system wide, and it does not disable swapping of the caller. (See SetIntHandler.)

A 386 trap handler is different from a 286 trap handler in that the access byte in the IDT (interrupt descriptor table) specifes a 386 trap gate rather than a 286 trap gate. A 386 trap gate directs the 386 processor to push additional information on the stack when the trap occurs. (See the *80386 Programmer's Reference Manual* for more information.)

Procedural Interface

Set386TrapHandler (iTrap, pTrapHandler): ercType

where

iTrap

is the trap type (0 to 255)

pTrapHandler

is the entry point of the trap handler.

Request Block

fRaw and *f386* are always 255.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	12
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	322
12	iTrap	2	
14	pTrapHandler	4	
18	reserved	4	
22	fRaw	1	255
23	f386	1	255

SetAlphaColorDefault (bMode): ercType

Caution: *This operation works on workstation hardware only. Do not use it on shared resource processor hardware.*

NOTE: This operation is supported by protected mode operating systems only.

Description

SetAlphaColorDefault changes control from the alpha or graphics palette currently in use to the default alpha palette and control structure. (See the *CTOS Programming Guide* for more information on using color in application programs.)

SetAlphaColorDefault creates the following alpha palette before calling ProgramColorMapper:

Palette Index	Color	Bit Pattern	Color
0	00Ch	00001100b	Green
1	008h	00001000b	Half-bright green
2	00Fh	00001111b	Cyan
3	00Ah	00001010b	Half-bright cyan
4	03Fh	00111111b	White
5	02Ah	00101010b	Half-bright white
6	03Ch	00111100b	Yellow
7	030h	00110000b	Red

Procedural Interface

SetAlphaColorDefault (bMode): ercType

where

bMode

is a byte value that is 0 to turn off graphics. If *bMode* is not 0, status code 7661 ("Invalid parameter to SetAlphaColorDefault") is returned.

Request Block

SetAlphaColorDefault is an object module procedure.

SetBsLfa (pBswa, lfa): ercType

Description

SetBsLfa sets the logical file address at which the input/output operation is to continue for the open file byte stream identified by the memory address of the Byte Stream Work Area. If the *lfa* contains 0FFFFFFFFh, the lfa of the file byte stream is set to the end-of-file *lfa* of the file. After setting the *lfa* to the end-of-file, the GetBsLfa operation can be called to determine the length of the file.

SetBsLfa is only valid for file byte streams; otherwise it returns status code 7 ("Not implemented").

Procedural Interface

SetBsLfa (pBswa, lfa): ercType

where

pBswa

is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

lfa

is the byte offset, from the beginning of the file, of the next byte to be read/written.

Request Block

SetBsLfa is an object module procedure.

This page intentionally left blank

SetDaBufferMode (pDawa, mode): ercType

Description

SetDaBufferMode sets the buffer management mode to write-through or write-behind.

Procedural Interface

SetDaBufferMode (pDawa, mode): ercType

where

pDawa

is the memory address of the same Direct Access Work Area that was supplied to OpenDaFile.

mode

is either the write-through or write-behind buffer management mode. This is indicated by 16-bit values representing the ASCII constants "wt" (write-through) and "wb" (write-behind). In these ASCII constants, the first character (w) is the high-order byte and the second character (t or b, respectively) is the low-order byte.

Request Block

SetDaBufferMode is an object module procedure.

This page intentionally left blank

SetDateTime (seconds, dayTimes2): ercType

Description

SetDateTime sets the current date and time in the System Date/Time format. (The System Date/Time format is described in "Utility Operations" in the *CTOS Operating System Concepts Manual*.)

Procedural Interface

SetDateTime (seconds, dayTimes2): ercType

where

seconds

is the count (0-43199) of seconds since the last midnight/noon.

dayTimes2

is the count (0-65535) of 12-hour periods since March 1, 1952.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	51
12	seconds	2	
14	dayTimes2	2	

SetDateTimeMode (modeDate, modeTime, modeFormat): ercType

Description

SetDateTimeMode changes the format in which the system date/time is displayed. At reboot, however, the date and time display reverts to the default format.

Procedural Interface

SetDateTimeMode (modeDate, modeTime, modeFormat): ercType

where

modeDate

is a byte value that determines the order in which the year (YY), month (MM), and day (DD) are displayed. The allowed values are

Value	**Description**
0	MM/DD/YY
1	DD/MM/YY
2	YY/MM/DD

modeTime

is a byte value that selects the standard AM/PM or the 24–hour clock format. The allowed values are

Value	**Description**
0	AM/PM format, for example 1:32 PM
1	24-hour clock format, for example 13:32

modeFormat

is a byte value that determines the display of the year, month, and day. The allowed values and example formats displayed by each are

Value	**Example Format**
0	Tue Jul 17, 1982
1	Tue 07/17/82
2	same as format for 0 value but columns are aligned

Request Block

SetDateTimeMode is an object module procedure.

SetDefault386TrapHandler (iTrap, pTrapHandler, ppTrapHandlerRet): ercType

Caution: *SetDefault386TrapHandler is supported in protected mode only on operating systems running on the 80386 microprocessor.*

Description

SetDefault386TrapHandler replaces the default system trap handler with a user-written system trap handler to handle the specified software generated interrupt (trap). The default and user-written handlers expect a 386 trap gate. (To replace a system default system handler that expects a 286 trap gate, see SetDefaultTrapHandler.)

The currently supported system default handlers are as follows:

Identifier	**Description**
0	Divide error
4	Overflow (INTO instruction)
5	Bounds check (BOUND instruction)
6	Invalid opcode
7	Coprocessor not available
11	Segment not present
13	General protection fault

A system default trap handler is accessible by any program that has not established a local trap handler using SetTrapHandler or Set386TrapHandler.

SetDefault386TrapHandler succeeds only if the caller is a protected mode, GDT-based program. On CTOS II 3.3 and higher operating systems, a trap may be specified for any of the interrupt levels (0 to 255) except level 14 and 205, which are reserved for operating system use.

SetDefault386TrapHandler returns the address of the default trap handler being replaced. A system service that calls SetDefault386TrapHandler may restore the original default trap handler during deinstallation.

Procedural Interface

SetDefault386TrapHandler (iTrap, pTrapHandler, ppTrapHandlerRet): ercType

where

iTrap

is the interrupt level (0 to 255) identifying the trap.

pTrapHandler

is the memory address of the user-written system trap handler.

ppTrapHandlerRet

is the memory address of a 4-byte memory area where the address of the trap handler being replaced is returned.

Request Block

cbTrapHandlerRet is always 4. *fRaw* and *f386* are always 255.

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	12
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	442
12	iTrap	2	
14	pTrapHandler	4	
18	reserved	4	
22	fRaw	1	255
23	f386	1	255
24	ppTrapHandlerRet	4	
28	cbTrapHandlerRet	2	4

This page intentionally left blank.

SetDefaultTrapHandler (iTrap, pTrapHandler, ppTrapHandlerRet): ercType

Description

SetDefaultTrapHandler establishes a different default trap handler in place of the system default trap handler that handles a processor-defined program exception. The currently supported defaults are as follows:

Identifier	**Description**
0	Divide error
4	Overflow (INTO instruction)
5	Bounds check (BOUND instruction)
6	Invalid opcode
7	Coprocessor not available
11	Segment not present

A system default trap handler is accessible by any program that has not established a local trap handler using SetTrapHandler or Set386TrapHandler.

SetDefaultTrapHandler succeeds only if the calling program is protected-mode/GDT-based. With CTOS II 3.3, a trap can be associated with any of the interrupt levels except 14 and 205.

SetDefaultTrapHandler returns the address of the default trap handler being replaced. A system service that calls SetDefaultTrapHandler may restore the original default trap handler during deinstallation.

Procedural Interface

SetDefaultTrapHandler (iTrap, pTrapHandler, ppTrapHandlerRet): ercType

where

iTrap

identifies the trap.

pTrapHandler

is the memory address of a GDT-based trap handler.

ppTrapHandlerRet

is the memory address of a 4-byte memory area where the address of the trap handler being replaced is returned.

Request Block

cbTrapHandlerRet is always 4. *fRaw* is always 255. *f386* is always 0.

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	12
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	339
12	iTrap	2	
14	pTrapHandler	4	
18	reserved	4	
22	fRaw	1	255
23	f386	1	0
24	ppTrapHandlerRet	4	
28	cbTrapHandlerRet	2	4

SetDeltaPriority (userNum, deltaPriority): ercType

Description

The SetDeltaPriority primitive sets a value that the Kernel adds to the priority of a process before determining which process to dispatch. This value, the "delta" priority, is the same for all processes with a given user number. SetDeltaPriority can change the effect of the priority parameter of any operation that has priority as a parameter.

SetDeltaPriority should not be used by programs that run under a partition-managing program, such as the Context Manager.

This operation is used to ensure that the processes with which a user is currently interacting have a higher priority than other processes. The delta priority is preserved across the Chain, Exit, and ChangePriority operations. SetDeltaPriority operates only if the priorities before and after its operation are in the range of 65 through 254.

Procedural Interface

SetDeltaPriority (userNum, deltaPriority): ercType

where

userNum

specifies the user number of all the processes for which the delta priority is to be altered.

deltaPriority

is the new value of the delta priority.

Request Block

SetDeltaPriority is a Kernel primitive.

This page intentionally left blank

SetDeviceHandler (tyDev, pIntHandler, saData, fRaw): ercType

Description

SetDeviceHandler, like the SetIntHandler operation, establishes a raw or mediated interrupt handler for device-generated interrupts. The description of SetDeviceHandler is essentially the same as that of SetIntHandler. (See the description of SetIntHandler.) The procedural interface, however, differs in the following ways:

- With SetDeviceHandler, the caller specifies the device type (*tyDev*) instead of the interrupt level (*iInt*) as the first parameter. The operating system determines the interrupt level from the value of *tyDev*. (For a list of the device types and their values, see the description of the ControlInterrupt operation.)

- With SetIntHandler, there is one additional flag parameter (*fDeviceInt*) that is TRUE if the interrupt handler serves device-generated interrupts. Because the interrupt handler established by SetDeviceHandler serves device-generated interrupts, this flag is not needed.

Procedural Interface

SetDeviceHandler (tyDev, pIntHandler, saData, fRaw): ercType

where

tyDev

is the device type (word). For a list of the device types and their values, see the description of the ControlInterrupt operation.

pIntHandler

is the memory address (entry point) of the interrupt handler.

saData

is the segment base address (word) of the data segment used by the mediated interrupt handler (mediated interrupt handlers only).

fRaw

is a flag (byte) that is TRUE if the device handler serves raw interrupts. Otherwise the device handler serves mediated interrupts.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	12
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	310
12	tyDev	2	
14	pIntHandler	4	
18	saData	2	
20	reserved	2	
22	fRaw	2	

SetDevParams (pbDevSpec, cbDevSpec, pbPassword, cbPassword, paramCode): ercType

Description

SetDevParams allows the characteristics of the floppy disk controller to be modified to accommodate various floppy disk formats.

Procedural Interface

SetDevParams (pbDevSpec, cbDevSpec, pbPassword, cbPassword, paramCode): ercType

where

pbDevSpec
cbDevSpec

describe a character string of the form {Node}[DevName]. Square brackets are optional for the device name. The distinction between uppercase and lowercase is not significant in matching device names.

pbPassword
cbPassword

describe the device password that authorizes access to the specified device.

paramCode

is a word that describes the desired characteristics to which the floppy disk controller is to be initialized. The codes are as follows:

Code	Density	Sector Size	Compatibility
0	single	128	IBM Diskette 1
1	single	256	IBM Diskette 2
2	single	512	---
3	double	256	IBM Diskette 2D
4	double	512	CTOS 8-inch IWS
5	double	1024	IBM Diskette 2D
6	---	---	reserved
7	double	256	CTOS (single-sided 5-1/4-inch floppy)
8	double	256	CTOS (double-sided 5-1/4-inch floppy)
9	double	512	Single-sided 5-1/4-inch floppy with 8 sectors per track (48 tracks per inch)
10	double	512	Double-sided 5-1/4-inch floppy with 8 sectors per track (48 tracks per inch)
11	double	512	Same as 9 above, but 9 sectors per track
12	double	512	Same as 10 above, but 9 sectors per track
13	double	512	Double-sided 5-1/4-inch floppy with 8 sectors per track (96 tracks per inch)
14	double	512	Same as 13 above, but 9 sectors per track
15	high capacity	512	Double-sided 5-1/4-inch (1.4mb) floppy with 16 sectors per track (96 tracks per inch)
16	high capacity	512	IBM 5-1/4-inch (1.2 mb) floppy with 15 sectors per track

Code	Density	Sector Size	Compatibility
17	high capacity	512	IBM 3-1/2-inch (1.44 mb) floppy
18	double	512	IBM 3-1/2-inch (760K byte) floppy

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	16
12	paramCode	2	
14	reserved	4	
18	pbDevSpec	4	
22	cbDevSpec	2	
24	pbPassword	4	
28	cbPassword	2	

This page intentionally left blank

SetDirStatus (pbDirName, cbDirName, pbPassword, cbPassword, statusCode, pStatus, sStatus) : ercType

Description

SetDirStatus allows a user to change a directory password or default file protection level. A volume or directory password is required.

Status code 218 ("Bad mode") is returned if the statusCode is invalid.

Procedural Interface

SetDirStatus (pbDirName, cbDirName, pbPassword, cbPassword, statusCode, pStatus, sStatus) : ercType

where

pbDirName
cbDirName

describe the directory name.

pbPassword
cbPassword

describe a password that gives access to the directory.

statusCode

specifies one of the following:

Code	**Item**	**Size (Bytes)**
0	invalid	
1	invalid	
2	File Protection Level	2
3	Password	*

*The length of the password is defined by *sStatus*, which must be less than or equal to 12.

pStatus
sStatus

describe the memory area from which the status information is copied.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	3
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	217
12	statusCode	2	
14	reserved	4	
18	pbDirName	4	
22	cbDirName	2	
24	pbPassword	4	
28	cbPassword	2	
30	pStatus	4	
32	sStatus	2	

This page intentionally left blank

SetDiskGeometry (pbDevName, cbDevName, pbPassword, cbPassword, pDiskGeometry, sDiskGeometry): ercType

Description

SetDiskGeometry sets the disk geometry. A subsequent call to QueryDiskGeometry allows the caller to query the disk geometry set by this operation.

Procedural Interface

SetDiskGeometry (pbDevName, cbDevName, pbPassword, cbPassword, pDiskGeometryRet, sDiskGeometryRet): ercType

where

pbDevName
cbDevName

describe a character string containing the device or volume name. Square brackets surrounding the name are optional. The distinction between uppercase and lowercase is not significant in matching device names.

pbDevicePassword
cbDevicePassword

describe the device or volume password.

pDiskGeometry
sDiskGeometry

describe the memory area where the disk geometry information is placed. The format of the information is shown below:

Offset	Field	Size (bytes)	Description
0	cylindersPerDisk	2	are the number of cylinders per disk
2	tracksPerCylinder	1	are the number of tracks per cylinder
3	sectors/Track	1	are the number of sectors per track
4	bytesPerSector	2	are the number of bytes per disk sector
6	fRemovable	1	TRUE means the medium may be removed
7	fMountable	1	TRUE means the device is capable of having a valid CTOS file system
8	fUtilizeECC	1	TRUE means the device is configured to use error correction code to rectify read errors
9	fECCCapable	1	TRUE means the device is capable of using error correction code
10	fWritePrecompensation	1	TRUE means write precompensation is utilized by the disk
11	reserved	1	
12	sbResourceMgr	13	is an sb string containing the name of the service managing the disk
25	rgResourceId	5	is a array of 5 one-byte elements that contain physical address information for the disk
30	writePreCompCyl	2	is the starting cylinder for write precompensation current

Offset	Field	Size (bytes)	Description
32	gapLength	1	is the length of the unused area between sectors
33	stepRate	1	is an encoded number that represents the time interval between seek step pulses for disks that are controlled by a WD-1010 or WD-2010
34	spiralFactor	1	is the sector number offset between adjacent tracks for disks (typically SMD) that use a spiral layout of data sectors to improve performance
35	interleave	1	
36	blocksPerDisk	4	are the number of user addressable data blocks (usually 512 bytes) per disk

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	3
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	367
12	reserved	6	
14	pbDevName	4	
18	cbDevName	2	
14	pbDevPassword	4	
18	cbDevPassword	2	
14	pDiskGeometry	4	
18	sDiskGeometry	2	

SetDispMsw287 (wMsw): ercType

Description

SetDispMsw287 directs the Kernel to set the machine status word (MSW) of the 80286/80386 microprocessors on every process context switch. This facility is of use to software that manages or emulates the numeric coprocessor.

SetDispMsw287 can set the task switch (TS), emulate coprocessor (EM), and monitor coprocessor (MP) bits but cannot set the protection enable (PE) bit.

The operating system maintains separate MSWs for real mode and protected mode.

If SetDispMsw287 is called by a real mode program, only the real mode MSW is set.

If SetDispMsw287 is called by a protected mode program, only the protected mode MSW is set. However, if the 8000h bit is set in the MSW, only the real mode MSW is set.

Procedural Interface

SetDispMsw287 (wMsw): ercType

where

wMsw

is the MSW.

Request Block

SetDispMsw287 is a Kernel primitive.

SetExitRunFile (pbExitRunFile, cbExitRunFile, pbPassword, cbPassword, priority): ercType

Description

SetExitRunFile establishes a new exit run file for the application partition in which the calling process is executing. If the current partition does not have an Application System Control Block, it is a system partition and the specified run file becomes the global exit run file.

Procedural Interface

SetExitRunFile (pbExitRunFile, cbExitRunFile, pbPassword, cbPassword, priority): ercType

where

pbExitRunFile
cbExitRunFile

describe a character string of the following form: {Node}[VolName]<DirName>FileName that specifies the run file to be loaded into the application partition when an exit request is issued by the current program.

pbPassword
cbPassword

describe the volume, directory, or file password that authorizes access to the specified file.

priority

is the priority at which the newly created process is scheduled for execution. The highest priority is 0; the lowest is 254.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqcode	2	186
12	priority	2	
14	reserved	4	
18	pbExitRunFile	4	
22	cbExitRunFile	2	
24	pbPassword	4	
28	cbPassword	2	

SetFhLongevity (fh, fLongLived): ercType

Description

SetFhLongevity sets a flag (*fLongLived*) to indicate how long a file handle is to survive. If *fLongLived* is FALSE, the file handle is marked short-lived (the default condition when a file is first opened), and the file is closed by the CloseAllFiles, Exit, ErrorExit, and Chain operations. If *fLongLived* is TRUE, the file handle is marked long-lived, and the file is closed only by an explicit CloseFile operation or by a CloseAllFilesLL operation.

Procedural Interface

SetFhLongevity (fh, fLongLived): ercType

where

fh

is the file handle returned from an OpenFile operation. The file can be opened in either read or modify mode.

fLongLived

is either FALSE for a short-lived file handle or TRUE for a long-lived one.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	30
12	fh	2	
14	fLongLived	1	

SetField (fh, pBuffer, sBuffer, pbFieldName, cbFieldName, pbValue, cbValue): ercType

Description

SetField modifies a file by searching for a ":FieldName:" string entry and updating the string value found following the trailing colon of the entry. If the entry is not found, the ":FieldName:" string as well as the specified value are appended to the file.

The search begins at the current location in the file. If the field is unique, it is recommended that the LookUpReset operation be called before this operation is used.

Procedural Interface

SetField (fh, pBuffer, sBuffer, pbFieldName, cbFieldName, pbValue, cbValue): ercType

where

fh

is the file handle (word) of the file to be modified.

pBuffer
sBuffer

describe a buffer to be used in the search. *sBuffer* (word) must be a multiple of 512.

pbFieldName
cbFieldName

describe the field name entry to be searched for. The colons preceding and following the entry should not be given as part of the field name.

pbValue
cbValue

describe the memory area where the string following the field name trailing colon is copied.

Request Block

SetField is an object module procedure.

SetFieldNumber (fh, pBuffer, sBuffer, pbFieldName, cbFieldName, w): ercType

Description

SetFieldNumber modifies a file by searching for a ":FieldName:" string entry and updating the string value found following the trailing colon of the entry with the specified number translated into a string. If the entry is not found, the ":FieldName:" string as well as the translated number are appended to the file.

Procedural Interface

SetFieldNumber (fh, pBuffer, sBuffer, pbFieldName, cbFieldName, w): ercType

where

fh

is the file handle (word) of the file to be modified.

pBuffer
sBuffer

describe a buffer to be used in the search. *sBuffer* (word) must be a multiple of 512.

pbFieldName
cbFieldName

describe the field name entry to be searched for. The colons preceding and following the entry should not be given as part of the field name.

w

is the number (word) to be translated into string format (ASCII) and inserted following the field name trailing colon.

Request Block

SetFieldNumber is an object module procedure.

SetFileStatus (fh, statusCode, pStatus, sStatus): ercType

Description

SetFileStatus copies the specified status information from the specified memory area to the File Header Block for the file defined by the file handle. SetFileStatus cannot change the file length. The ChangeFileLength operation can be used to change the file length.

Procedural Interface

SetFileStatus (fh, statusCode, pStatus, sStatus): ercType

where

fh

is a file handle returned from an OpenFile operation. Except when setting the cache control status (*statusCode* 13) or when setting the file removal status (*statusCode* 14), the file must be open in modify mode. When setting *statusCode* 13, the file can be opened in read or modify mode. In read mode caching of the specified file or device is affected only for the duration of this file or device open. In modify mode, the caching status set by this operation persists across all opens and affects all clients that access the file or device. When setting *statusCode* 14, the file must be opened with a password that permits reading and modifying the file.

statusCode

specifies the status. Status items and their codes are as follows:

Code	**Item**	**Size (Bytes)**
1	File type	1
2	File protection level	1
3	Password	*
4	Date/time of creation	4
5	Date/time last modified	4
6	End-of-file pointer	4
7	reserved	
8	reserved	
9	reserved	
10	Application-specific** field in the File Header Block	64
11	reserved	
12	File Header Block+	512
13	Cache control status	2
14	File removal status	1
15	Internationalization	2

* The length of password is defined by sStatus, which must be less than or equal to 12.

**This field is used by Word Processor files, for example.

+ SetFileStatus sets certain fields in the File Header Block. These are listed below.

Code 12, File Header Block, sets the following fields in the File Header Block structure:

Offset	Item	Size (Bytes)
55	sbPassword	13
86	fileClass	1
87	accessProtection	1
92	creationDT	4
96	modificationDT	4
100	accessDT	4
104	expirationDT	4
108	fNoSave	1
109	fNoDirPrint	1
111	lfaEndOfFile	4
115	defaultExpansion	4
377	(reserved for CT expansion)	71
448	application-specific field	64

Code 13, cache control status, indicates whether accesses to the file or device specified by *fh* will use the system file cache. The bits in the word of control information returned have the following meanings:

Bit	Field	Description
0	mCacheDisable	Do not use the system file cache in accessing this file.
1	mCacheEnable	Use the system file cache in accessing this file. Asserting *mCacheEnable* and *mCacheDisable* is mutually exclusive. Only one bit may be set at a time. If neither is set, cacheability defaults to the state defined by the system default.

The cache-enabled state established by code 13 overrides the system default established at initialization through the system configuration file. For example, if the system default is that caching be enabled, all disk accesses will use the cache unless SetFileStatus is invoked to turn off caching for selected files or devices. If caching has been disabled for a device, accesses to files on that device will not use the file cache unless SetFileStatus is invoked to turn on caching for selected files. This field is valid only on systems that support the file cacher. (See Appendix A, "Operating System Features," in the *CTOS Operating System Concepts Manual.*)

Code 14, file removal status, indicates whether a file should be deleted when its file handle is closed. If the value at the location pointed to by *pStatus* is 1, SetFileStatus causes the file to be removed from the directory when the indicated file handle is closed. When all the file handles are closed, a value of 1 causes the file to be removed from the disk. However, the file is only affected if the file handles are closed by CloseFile. (For details, see "Deleting a File," in the "File Management" section of the *CTOS Operating System Concepts Manual*)

Code 15, internationalization, sets the *wInternational* field in the File Header Block.

pStatus
sStatus

describe the memory area from which the status information is copied.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	9
12	fh	2	
14	statusCode	2	
16	reserved	2	
18	pStatus	4	
22	sStatus	2	

This page intentionally left blank

SetIBusHandler (wId, pProc): ercType

Caution: *This operation should be used only by programs executing in protected mode.*

Description

SetIBusHandler allows the caller to register (with an I-Bus interrupt handler) a procedure for handling or processing data from a specific I-Bus device. The ResetIBusHandler operation performs the reverse effect of this operation by disconnecting an I-Bus handler from a specific I-Bus device. (See ResetIBusHandler.)

NOTE: The memory address of the procedure to be registered must be based in the Global Descriptor Table (GDT).

Procedural Interface

SetIBusHandler (wId, pProc): ercType

where

wId

is the 2 byte I–Bus device identifier or 0FFxxh, where xx is the device attribute byte (lower byte) of the device identifier. When 0FFxxh is used (such as 0FFB0h for example), the procedure at *pProc* (described below) receives data from all devices with xxh as the lower byte of their identifier. In this case, the value of the device identifier upper byte is ignored.

pProc

is the memory address of the procedure. Typically, this procedure is part of the service that handles the specific I-Bus device. The procedure is called from an interrupt level with a byte of data from the I–Bus device.

The I–Bus device handler is called with the following procedural interface:

wRetCode =pProc (bIn, prgbEventCode, prgbCharCode, pbEntries,
pfQueue, fReset)

where

bIn

is the byte received from the I–Bus device.

prgbEventCode

is the memory address of an array into which the installed I-Bus handler can place event codes. Each event code must have a corresponding character in the character code array, *rgbCharCode*. Event codes are queued for access by the ReadInputEvent operation only if the flag *fQueue* is TRUE. (See the description of ReadInputEvent in this chapter.)

prgbCharCode

is the memory address of an array into which the installed I-Bus handler can place character codes. Each character code must have a corresponding event code in the array of event codes, *rgbEventCode*. Character codes are queued for access by the ReadInputEvent only if the flag *fQueue* is TRUE.

pbEntries

is the memory address of a byte into which the number of I-Bus handler event code/character code pairs is placed. If *bEntries* is 0 (or if *fQueue* is FALSE), no events are queued for access by ReadInputEvent.

pFQueue

is the memory address of a flag (byte) into which the I-Bus handler must place TRUE or FALSE. TRUE causes *bEntries* (may be 0) keyboard events to be queued. TRUE also finishes processing of the I-Bus byte: *bIn* will not be passed to other I-Bus handlers serving the same device. FALSE queues no keyboard events, and other I-Bus handlers (if any) are called.

fReset

is normally FALSE when the procedure at *pProc* is called. If *fReset* is TRUE, the I-Bus device responded with a 2 byte device identifier. In such a case, *bIn* is the higher byte of the identifier.

wRetCode

is a word that determines how the next character from the I-Bus device is to be interpreted. The returned values must be either 0 or 1. A value of 0 informs the operating system to process I-Bus protocol normally. A value of 1 notifies the operating system to call the handler on the next character, even if the character is a protocol character.

The I–Bus device handler usually is part of a system service designed to handle data from a specific device, such as a card reader. In this case, the handler processes the device data and may elect not to set *fQueue*, since applications may not be expected to access the card reader data by calling ReadInputEvent. If the handler does set *fQueue* (to queue events and character codes), these events are available to ReadInputEvent callers. If *fQueue* is TRUE and the event code indicates that the device is a keyboard, the event is queued for access by the ReadKbd, ReadKbdDirect, and ReadKbdDataDirect operations. If, however, the event code indicates the device is a pointing device (Mouse attribute is 0), the event is queued for access by the Mouse Sevice.

By using the event queueing mechanism, an I–Bus handler can be used to filter I–Bus data to the keyboard or mouse, for example. In fact, if *fQueue* is TRUE and *bEntries* is 0, a handler effectively can disable an I-Bus device by intercepting data from the device but not queueing the data (*bEntries* is 0).

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	348
12	wID	2	
14	pProc	4	

SetImageMode (pBswa, mode): ercType

Description

SetImageMode sets the normal, image, or binary mode for printer, spooler, and communications byte streams. SetImageMode, if attempted on other byte streams, returns status code 7 ("Not implemented").

Procedural Interface

SetImageMode (pBswa, mode): ercType

where

pBswa

is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

mode

is a code as follows:

Code	**Description**
0	for normal mode
1	for image mode
2	for binary mode

Request Block

SetImageMode is an object module procedure.

This page intentionally left blank

SetImageModeC (pBswa, mode): ercType

Description

SetImageModeC is the version of SetImageMode that is device-specific to RS-232 serial ports ([Comm] and [Ptr] byte streams).

(See "Device-Dependent SAM" and "Communications Programming" in the *CTOS Operating System Concepts Manual* for more information.)

Procedural Interface

SetImageModeC (pBswa, mode): ercType

where

pBswa

is the memory address of the Byte Stream Work Area (BSWA) as given to the OpenByteStream call.

mode

is one of the following values:

Value	**Description**
0	normal
1	image
2	image-binary

Request Block

SetImageModeC is an object module procedure.

This page intentionally left blank

SetIntHandler (iInt, pIntHandler, saData, fDeviceInt, fRaw): ercType

Description

SetIntHandler establishes a raw or mediated interrupt handler. When an application program terminates, its interrupt handler is detached and the default interrupt handler again serves the interrupts.

Unlike Set386TrapHandler or SetTrapHandler, SetIntHandler disables swapping of the caller and is invoked when any process, regardless of its user number, generates the interrupt.

On protected mode systems, SetIntHandler returns an error if the specified interrupt level is not currently serving a hardware generated interrupt.

For device-generated interrupts, SetDeviceHandler should be used when the interrupt level for a device varies with the type of hardware. For example, the interrupt level for the SCSI hard disk could be at level 67 on one machine and level 86 on another. SetDeviceHandler is machine-independent because it takes a device type parameter instead of an interrupt level. The device type is mapped to the proper interrupt level depending on the hardware. (See SetDeviceHandler.)

Procedural Interface

SetIntHandler (iInt, pIntHandler, saData, fDeviceInt, fRaw): ercType

where

iInt

is the interrupt level (a word value in the range of 0 to 255).

pIntHandler

is the memory address of the interrupt handler entry point.

saData

is the segment base address of the data segment that is used by the mediated interrupt handler (for mediated interrupt handlers only).

fDeviceInt

is a flag (byte). TRUE indicates the interrupt handler serves device-generated interrupts. FALSE indicates the interrupt handler serves software-generated interrupts (for mediated interrupt handlers only).

fRaw

is a flag (byte). TRUE indicates the interrupt handler serves raw interrupts. FALSE indicates the interrupt handler serves mediated interrupts.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	12
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	69
12	iInt	2	
14	pIntHandler	4	
18	saData	2	
20	fDeviceInt	2	
22	fRaw	2	

This page intentionally left blank

SetIOOwner (pBuffer, sBuffer, piRecordRet): ercType

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

SetIOOwner acquires or releases exclusive access rights to one or more contiguous ranges of input/output addresses. If any single address in a specified range is owned by another process, SetIOOwner returns a status code to the calling process and does not acquire or release any addresses in the range.

The application provides a record of each range of addresses for which ownership is to be changed. SetIOOwner changes address ownership in the order in which the corresponding records at *pBuffer* appear.

Before an application calls SetIOOwner, it can call QueryIOOwner to identify addresses that are currently owned by other processes. (See QueryIOOwner.)

SetIOOwner is primarily used by the PC Emulator to prevent multiple processes from simultaneously using the same I/O addresses.

Status code 434 ("Address already owned") is returned if one or more addresses in a range is owned by another process. In addition, the index number of the offending record is returned at *piRecordRet*.

Procedural Interface

SetIOOwner (pBuffer, sBuffer, piRecordRet): ercType where

pBuffer

is the memory address of a buffer that contains one or more records. Each record has the following format:

Offset	Field	Size (Bytes)	Description
0	qFirstIoAddr	4	The first address in the range of contiguous input/output addresses.
4	cIoAddr	4	The count of addresses in the range of input/output addresses. Each address corresponds to one byte in memory.
8	fAcquire	1	A flag. TRUE acquires ownership of the range of addresses. FALSE releases ownership of the range of addresses.
9	reserved	1	

sBuffer

is the size, in bytes, of the buffer at *pBuffer*.

piRecordRet

is the memory address of a word where the index number of a record that caused an error is returned. Note that the first record has an index number of 0.

Request Block

siRecordRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	464
12	reserved	6	
18	pBuffer	4	
22	sBuffer	2	
24	piRecordRet	4	
28	siRecordRet	2	2

This page intentionally left blank.

SetKbdLed (iLed, fOn): ercType

Caution: *This operation works on workstation hardware only. Do not use it on shared resource processor hardware.*

Description

SetKbdLed turns keyboard LEDs on or off.

Procedural Interface

SetKbdLed (iLed, fOn): ercType

where

iLed

is the identification (word) of the LED to turn on/off as follows:

iLED	Key
0	f10
1	f9
2	f8
3	f3
4	f2
5	f1
6	LOCK (only if the keyboard is in unencoded mode)

iLED	Key
7	OVERTYPE
8	LTAI
9	ENQ
10	LOCAL
11	RCB
12	XMT
13	num Lock
14	CTOS Lock

NOTE: Bits 8 through 12 are only valid on K3 and K5 keyboards. Bits 13 and 14 are only valid on the SG-102-K keyboard.

fOn

is a flag (word) that is TRUE to turn the LED on and FALSE to turn it off. Alternately, if *iLed* is FFFFh, the status of each bit in *fOn* determines the state of each LED. In this case, each bit number in *fOn* corresponds to the identification number of an LED. For example, if *iLed* is FFFFh and bit 3 in *fOn* is set, the LED on the **F3** key is turned on. If bit 3 is cleared, the LED on the **F3** key is turned off.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	56
12	iLed	2	
14	fOn	2	

This page intentionally left blank

SetKbdUnencodedMode (fOn): ercType

Caution: *This operation works on workstation hardware only. Do not use it on shared resource processor hardware.*

Description

SetKbdUnencodedMode selects the keyboard mode. SetKbdUnencodedMode discards the contents of the type-ahead buffer if the mode is actually changed.

Modes 128 and 129 are only supported by the ReadKbdInfo and ReadInputEvent operations. (See ReadKbdInfo and ReadInputEvent.)

Procedural Interface

SetKbdUnencodedMode (fOn): ercType

where

fOn

is the mode (word) selected. The allowed values are

Value	**Description**
0	Character mode. This mode allows the application to receive character codes.
1	Mapped unencoded mode (K1 keyboard emulation). This mode allows the application to receive mapped unencoded values.

Value	Description
2	Raw unencoded mode. This mode allows the application to receive the unencoded values specific to the attached keyboard.
80h	Character plus mode. This mode allows the application to receive both the character code and the raw unencoded key values.
81h	Unencoded plus mode. This mode allows the application to receive both the mapped unencoded and the raw unencoded key values.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	57
12	fOn	2	

SetKeyboardOptions (mode, prgOptions, sOptions): ercType

NOTE: This operation is supported only on protected-mode operating systems.

Description

SetKeyboardOptions changes the keyboard configuration by optionally performing the functions listed below.

- It allows the caller to map a source keyboard to a target keyboard.
- It specifies whether the keyboard will beep when a **Lock** or **Num Lock** key is toggled.
- It stops keys from repeating or reduces the rate at which they repeat.
- It sets the chords or the **Action** key to remain in effect when released.
- It prevents keys from repeating when held down at the same time, with the exception of the last key pressed. Unlike the other functions, which affect all partitions in memory, this function only applies to the partition of the calling process.

Procedural Interface

SetKeyboardOptions (mode, prgOptions, sOptions): ercType

where

mode

is a value that is reserved for future use. It should always be set to 0.

prgOptions

is the memory address of an array of double words. The low-order word of each array element contains the option number. The high-order word contains the option setting. Each array element is described below.

NOTE: The fBeep and repeatRate options only work for applications reading the keyboard in character mode.

Option (Low-order word)	**Field (High-order word)**	**Description**
1	fChordStick	TRUE causes the chord to remain in effect when released until the next key is pressed.
2	fActionStick	TRUE causes the **Action** key to remain in effect when released until the next key is pressed.
3	fBeep	TRUE causes the keyboard to emit one beep followed by a short beep when the **Lock** key is turned on. TRUE also causes the keyboard to emit one beep followed by two short beeps when the **Lock** key is turned off.

Option (Low-order word)	Field (High-order word)	Description
4	repeatRate	Indicates how fast a key is to repeat when held down. Each value in the high-order word indicates the following: **Value** / **Description** 0 — Normal rate 1 — 50% of normal rate 2 — 25% of normal rate 3 or more — Disables repeating
1	fChordStick	TRUE causes the chord to remain in effect when released until the next key is pressed.
5	kbdIDMapping	Indicates the target keyboard ID (low-order byte) and the source keyboard ID (high-order byte).
6	fKeyRepeat	TRUE prevents a group of keys, with the exception of the last key, from repeating when held down at the same time.

sOptions

is the size of the array at *prgOptions*.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	448
12	mode	2	
14	reserved	4	should be 0
18	prgOptions	4	
22	sOptions	2	
24	pb	4	reserved
28	cb	2	reserved

SetLdtRDs (sgLdtr, sgDs): ercType

Description

SetLdtRDs sets the caller's Local Descriptor Table (LDT) and data segment (DS) selectors. To preserve the LDT selector across task switch, SetLdtRDs also updates the caller's Task State Segment (TSS) with the LDT selector.

This operation is used by a GDT-based caller before calling an LDT-based operation. For example, mediated interrupt handlers that make procedure calls to LDT-based code provided by a client may use SetLdtRDs to set up the client's LDT and DS selectors.

Procedural Interface

SetLdtRDs (sgLdtr, sgDs): ercType

where

sgLdtr

is the new LDT selector.

sgDs

is the new DS selector.

Request Block

SetLdtRDs is a system-common procedure.

This page intentionally left blank

SetLPIsr (pLpIsr, saData): ercType

Description

SetLPIsr establishes the printer interrupt service routine (PISR) to process interrupts generated by the parallel printer interface. A PISR established by an application program is reset automatically when the application program terminates.

Procedural Interface

SetLPIsr (pLpIsr, saData): ercType

where

pLpIsr

is the memory address (CS:IP) of the printer interrupt handler. If it is 0, it resets the current interrupt handler.

saData

is the value of the data segment (DS), which is used by the printer interrupt handler.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	121
12	saData	2	
14	pLpIsr	4	

SetModuleId (iBus, nPos, qModuleId): ercType

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

SetModuleId sets the identification number of the specified module/cartridge on the input device bus (I-Bus), system expansion bus (X-Bus), or SuperGen system bus (X-Bus+).

This interface is provided for system services that manage hardware devices and need to identify the device for other services and applications. For example, VAM uses this interface to set the module id for the video adapter. On EISA/ISA-Bus workstations, this is necessary because the video adapter is not self-identifying and the operating system has no video-specific knowledge.

For I-Bus and X-Bus modules, the high byte of the low word module ID (a word) indicates the module type. The low byte, in turn, indicates module attributes, such as the version number. (See Appendix G for a list of all the I-Bus and X-Bus modules and their IDs.)

For X-Bus+ cartridges, the low word of the ID (a double word) is the manufacturer's code, and the high word is the product ID. For a X-Bus module that has been redesigned as a X-Bus+ cartridge (and is 100% compatible with its X-Bus counterpart), the manufacturer's code is zero, and the high word is of the same format as the X-Bus module ID number.

If the current operating system version does not recognize the specified bus, SetModuleId returns status code 36 ("No such bus"). If the specified bus type does not exist on the present workstation, or if no modules/cartridges are connected to the specified bus, SetModuleId returns status code 35 ("No bus or module present").

Procedural Interface

SetModuleId (iBus, nPos, qModuleId): ercType

where

iBus

specifies the type of bus and is one of the following values:

Value	**Description**
1	I-Bus
2	X-Bus
3	X-Bus+

nPos

specifies the module/cartridge position on the bus. This index starts at 1.

qModuleId

is a 32-bit module id.

Request Block

SetModuleId is a system-common procedure.

SetMsgRet (pbMsg, cbMsg): ercType

Description

SetMsgRet places a message in the caller's long-lived memory. It also places the memory address of the message in the *pbMsgRet/cbMsgRet* field in the caller's Application System Control Block (ASCB). (See Chapter 4, "System Structures," for the format of the ASCB.) The exit run file, typically the Executive, will display the message when it is reloaded into memory.

A system service must call SetMsgRet rather than calling ErrorExitString during its installation if it wants to provide the exit run file with an informative message indicating the success or failure of installation. SetMsgRet must be called prior to calling ConvertToSys. (See "System Services Management" in the *CTOS Operating System Concepts Manual.*)

Procedural Interface

SetMsgRet (pbMsg, cbMsg): ercType

where

pbMsg
cbMsg

describe the message placed in the caller's long-lived memory.

Request Block

SetMsgRet is an object module procedure.

This page intentionally left blank

SetNode (pbNodeName, cbNodeName): ercType

Description

SetNode allows the specification of a node name to be used as part of the default path whenever a file specification is given that does not contain a node name or volume name.

Procedural Interface

SetNode (pbNodeName, cbNodeName): ercType

where

pbNodeName
cbNodeName

describe a node name.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	252
12	reserved	6	
18	pbNodeName	4	
22	cbNodeName	2	

SetPartitionExchange (exchange): ercType

NOTE: SetPartitionExchange is documented for backwards compatibility with older programs only. New applications should use the Intercontext Message Service (ICMS) rather than this operation for communicating with applications in other partitions. (For details, see "Interprocess Communication" in the CTOS Operating System Concepts Manual.)

Description

SetPartitionExchange sets up an exchange number that can be queried by a program in another application partition for interpartition communication.

After this call is made, the exchange persists until the operating system is rebooted. The caller cannot deallocate it with the DeallocExch operation.

Procedural Interface

SetPartitionExchange (exchange): ercType

where

exchange

is an exchange (word) previously allocated by the application program.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	183
12	exchange	2	

SetPartitionLock: (fLock) ercType

Description

SetPartitionLock declares whether an application program executing in the specified application partition is locked. If it is locked, it cannot be terminated by the TerminatePartitionTasks or VacatePartition operation.

An application program can lock itself into its own partition only. A partition containing a locked program can be swapped out of memory but it cannot be vacated.

An Exit or ErrorExit from an application program in the locked partition vacates the application partition, but no other run file is loaded and the partition cannot be deleted except by system reload.

Procedural Interface

SetPartitionLock (fLock): ercType

where

fLock

is a flag (byte). TRUE means that the partition is locked. FALSE means that the partition is unlocked.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	182
12	fLock	1	
13	reserved		

SetPartitionName (userNumPartition, pbName, cbName): ercType

Description

SetPartitionName changes the name of the caller's partition. Other programs can then use the operating system call GetPartitionHandle to determine whether a specific program is running.

Procedural Interface

SetPartitionName (userNumPartition, pbName, cbName): ercType

where

userNumPartition

is the user number (word) of the partition. The value 0 specifies the partition in which the client is executing.

pbName
cbName

describe the new partition name. The name must be 12 characters or less in length. The name must be unique.

Request Block

SetPartitionName is an object module procedure.

This page intentionally left blank

SetPartitionSwapMode (userNumPartition, wSwapMode): ercType

Description

SetPartitionSwapMode sets a swap mode for the specified partition. The swap mode determines the policy (specified by *wSwapMode*) used for swapping partition memory to disk when memory is oversubscribed.

This swap mode is long-lived, that is, the swap mode persists across all chain operations.

Procedural Interface

SetPartitionSwapMode (userNumPartition, wSwapMode): ercType

where

userNumPartition

> is the user number (word) returned from a CreatePartition or GetPartitionHandle operation. A 0 specifies the application partition in which the client is executing.

wSwapMode

is a word value that indicates the swapping policy for the partition indentified by userNumPartition. The following swap modesarre supported:

Value	**Description**
0	Compatibility mode. Protected mode operating systems use the default swapping policy. By default, the operating system automatically swaps in only those partitions that do not contain data structures associated with a context managing program. (A partition is under context management if the flag *fCCB* in the Partition Descriptor is TRUE. See "Partition Descriptor" in Chapter 4, "System Structures.")
1	Do not swap out. Protected mode operating systems do not swap out the partition. This swap mode differs from the SetSwapDisable operation in that SetSwapDisable is not long-lived.
2	Enable automatic swap in. Protected mode operating systems support automatically swapping in partitions at periodic time intervals. You may enable this feature by placing a :WakeupInterval: entry in [Sys]<Sys>Config.sys. (See the *CTOS System Administration Guide* for details on Config.sys options.) This swap mode overrides the default policy described above.
3–65535	Reserved.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	365
12	userNumPartition	2	
14	wSwapMode	2	

This page intentionally left blank

SetPath (pbVolSpec, cbVolSpec, pbDirName, cbDirName, pbPassword, cbPassword): ercType

Description

SetPath establishes a default volume, a default directory, and a default password. It also clears the default file prefix. A subsequent ClearPath operation clears the defaults.

Use the SetNode operation to set a node name.

Procedural Interface

SetPath (pbVolSpec, cbVolSpec, pbDirName, cbDirName, pbPassword, cbPassword): ercType

where

pbVolSpec
cbVolSpec

describe the default volume specification.

pbDirName
cbDirName

describe the default directory name.

pbPassword
cbPassword

describe the default password.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	3
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	1
12	reserved	6	
18	pbVolSpec	4	
22	cbVolSpec	2	
24	pbDirName	4	
28	cbDirName	2	
30	pbPassword	4	
34	cbPassword	2	

SetPrefix (pbPrefix, cbPrefix): ercType

Description

SetPrefix establishes a default file prefix that is prefixed to the file name part of a file specification if that file specification does not have an explicit volume name or directory name. A new SetPrefix overrides a previous SetPrefix. The default prefix established by SetPrefix can be removed by:

- another SetPrefix that specifies a null string
- the SetPath operation
- the ClearPath operation

Procedural Interface

SetPrefix (pbPrefix, cbPrefix): ercType

where

pbPrefix
cbPrefix

describe the character string that is to be used as a default file prefix.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	3
12	reserved	6	
18	pbPrefix	4	
22	cbPrefix	2	

SetPStructure (wStructCode, userNum, oField, pb, cb): ercType

Description

SetPStructure provides controlled write access to selected fields of certain system data structures that may legitimately be modified by user programs running in protected mode.

SetPStructure is used primarily by object module procedures in Ctos.lib to perform system-related functions. In most cases, it is unnecessary for a user program to use SetPStructure directly.

SetPStructure provides write access on a field-by-field basis. Only certain fields may be modified using SetPStructure, and the values placed in these fields may be validated by SetPStructure. Status code 7 ("No procedures to implement") is returned if *wStructCode* specifies an unsupported case or if oField specifies an unsupported field.

A separate call to SetPStructure must be used to set each field to be modified, except in the case of adjacent *pb/cb* fields, which are modified by a single call to SetPStructure. In this respect, SetPStructure is unlike its counterpart, GetPStructure, which returns the memory address of the entire system data structure.

Procedural Interface

SetPStructure (wStructCode, userNum, oField, pb, cb): ercType

where

wStructCode

is a word value that identifies the system data structure. This parameter is identical to the corresponding parameter of GetPStructure.

userNum

is the user number (word) returned from a CreatePartition or a GetPartitionHandle operation. If *userNum* is 0, then the user number of the calling program is used.

oField

is the offset, within the system data structure, of the field to be modified. This parameter must correspond to the first byte of the field (see below).

pb

a memory address. The meaning of this parameter depends on the type of field being addressed (see below).

cb

a count of bytes. The meaning of this parameter depends on the type of field being addressed (see below).

SetPStructure is designed to be extended to support access to additional fields and new system data structures as necessary, without introducing new procedural interfaces. Because of this generality, the *pb* and *cb* parameters are interpreted differently, as described below, depending on the type of field being modified:

Field Type	**Interpretation**
p	The system data structure field is a 32-bit *sa:ra* logical address. The *pb* parameter is the pointer value to be placed in the structure (not the address of the value). The *cb* parameter is ignored.

sa	The field is a 16-bit segment address (in real mode, a paragraph number; in protected mode, a selector). The *sa* portion of the *pb* parameter (the high-order 16 bits) replaces the field. The *ra* portion of *pb* (the low-order 16 bits), and *cb*, are ignored.
pb/cb	A *pb* and *cb* pair occur as adjacent, related fields in a system data structure. The *pb* and *cb* operands replace the fields. This is the only instance in which two fields are modified by a single call to SetPStructure. The *oField* parameter should be the offset of the *pb* field.
sb	The field takes a variable-length string value. The first byte of the field is the current length of the string (excluding the length byte itself). The *pb* parameter is the address of the new string value (without a length byte). The new string value is copied to the field beginning at the second byte of the field. The *cb* parameter is the length of the string value, to be placed in the first byte of the field; it may not exceed the maximum size of the field minus one.
other	The field is any other fixed-length field, including byte, word, dword, or larger fixed-length fields. The *pb* parameter is the address of the new value (not the value itself), and *cb* must exactly match the size of the field.

The following fields are supported by SetPStructure:

Code	Structure	Offset	Field	Size
3	Application System Control Block	10	pbMsgRet/ cbMsgRet	6
13	Event Control Block	2	pbQMailId/ cbQMailId	6
		6	sbNodeMail	13
32	Partition Descriptor	16	pbName/ cbName	6

Request Block

SetPStructure is a system-common procedure.

SetRsLfa (pRswa, lfa): ercType

Description

SetRsLfa sets the logical file address (lfa) where the next operation will occur for an open Record Sequential Access Method (RSAM) file. The RSAM file must be opened in read mode or status code 3605 ("Wrong mode") is returned. If the lfa for the RSAM file contains 0FFFFFFFFh, the lfa is set to the lfa end-of-file. After the lfa is set to the end-of-file, the GetRsLfa operation can be called to determine the file length. (See GetRsLfa.)

If an invalid lfa is given, status code 3608 ("Bad RSLfa") is returned, and the logical address of the file becomes undefined.

Procedural Interface

SetRsLfa (pRswa, lfa): ercType

where

pRswa

is the memory address of the same RSWA that was supplied to OpenRsFile.

lfaSet

is the new logical file address (lfa) to be set. The lfa is a file position in the range of positions from 0 to 0FFFFFFFFh.

Request Block

SetRsLfa is an object module procedure.

This page intentionally left blank

SetScreenVidAttr (iAttr, fOn): ercType

Description

SetScreenVidAttr sets and resets a specified screen attribute.

Note that some attributes have a different meaning on a color workstation than they have on a monochrome workstation. You can use the PutFrameAttrs operation for those bits that are affected.

Procedural Interface

SetScreenVidAttr (iAttr, fOn): ercType

where

iAttr

is a word value that identifies the screen attribute. The values are

Value	**Screen Attribute**
0	reverse video
1	video refresh
2	half-bright
255	internal use only

fOn

is a flag (word) that is TRUE to turn the specified screen attribute on and FALSE to turn it off.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	77
12	iAttr	2	
14	fOn	2	

SetScreenVidAttr (iAttr, fOn): ercType

Description

SetScreenVidAttr sets and resets a specified screen attribute.

Note that some attributes have a different meaning on a color workstation than they have on a monochrome workstation. You can use the PutFrameAttrs operation for those bits that are affected.

Procedural Interface

SetScreenVidAttr (iAttr, fOn): ercType

where

iAttr

is a word value that identifies the screen attribute. The values are

Value	**Screen Attribute**
0	reverse video
1	video refresh
2	half-bright

fOn

is a flag (word) that is TRUE to turn the specified screen attribute on and FALSE to turn it off.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	77
12	iAttr	2	
14	fOn	2	

SetSegmentAccess (sn, bAccess): ercType

Description

In protected mode, SetSegmentAccess sets the access mode of a code or data segment. It can be used to convert a code segment to a data segment, or vice versa, and/or to set or clear the "readable code segment" attribute or the "writable data segment" attribute.

SetSegmentAccess does not provide any control over the following fields of the access rights byte:

- present bit
- DPL field
- segment bit
- conforming code/expand-down data segment bit
- accessed bit

In protected mode, code segments are never writable and may always be executed. Data segments are always readable and may never be executed.

Initially, user code segments are readable; user data segments are writable.

In real mode, SetSegmentAccess performs no function.

If the *sn* parameter denotes an expand-down data segment, the bAccess parameter also must denote a data segment. Only an expand-up data segment may be converted to a code segment. Converting a code segment to a data segment always results in an expand-up segment.

Procedural Interface

SetSegmentAccess (sn, bAccess): ercType

where

sn

is the selector (word) of the segment whose access mode is to be changed. The selector must denote a segment descriptor.

bAccess

is the new access mode (byte), which replaces the current mode in the access rights byte of the segment descriptor. It must be one of the following decimal values:

Value	**Description**
0	data segment, read only
2	data segment, read/write
8	code segment, execute only
10	code segment, read/execute

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	285
12	sn	2	
14	bAccess	1	
16	reserved	3	

This page intentionally left blank

SetStyleRam (fGraphics)

Caution: *This operation works on workstation hardware only and is documented for compatibility with older programs. New programs should use ProgramColorMapper.*

Description

SetStyleRam sets a flag (*fGraphics*) to turn on the bits in the character attribute byte that enable color. If *fGraphics* is TRUE and the appropriate bits are set, the bits will be interpreted as color. If FALSE, the bits for color are ignored.

(See the *CTOS Programming Guide* for details on setting the character attribute bits and for information on how to use the color palette.)

Procedural Interface

SetStyleRam (fGraphics)

where

fGraphics

is a flag (byte). TRUE turns on color if the appropriate character attribute bits are set. FALSE causes the bits for color to be ignored.

Request Block

SetStyleRam is an object module procedure.

This page intentionally left blank

SetStyleRamEntry (iEntry, bVal)

Caution: *This operation works on workstation hardware only and is documented for compatibility with older programs. New programs should use ProgramColorMapper.*

Description

SetStyleRamEntry is used to modify a single one-byte entry in the graphics style RAM. The position of the one-byte entry within the set of attributes must be specified, as well as the value for the byte that will be modified.

Procedural Interface

SetStyleRamEntry (iEntry, bVal)

where

iEntry

is a word that specifies which one of the eight bytes is to be replaced. 0 to 7 are the valid values.

bVal

is a byte that specifies the new value for the selected entry.

Request Block

SetStyleRamEntry is an object module procedure.

This page intentionally left blank

SetSwapDisable (fDisable): ercType

Description

SetSwapDisable allows a program to specify whether or not it can be swapped.

Using SetSwapDisable may cause other programs that are swapped out of memory to be unable to be swapped back in (due to memory constraints) until the calling program terminates.

Procedural Interface

SetSwapDisable (fDisable): ercType

where

fDisable

is a flag (byte) that is TRUE if the calling program is not to be swapped to disk.

Request Block

SetSwapDisable is a system–common procedure.

This page intentionally left blank

SetSysInMode (iMode, fhSysIn): ercType

Description

SetSysInMode changes the state of the System Input Process. This operation is used either to record input from the keyboard to a file or to play back input read from a file.

SetSysInMode automatically closes the file identified by *fhSysIn*

- at the end of a playback session when all of the characters in the file are read
- or when SetSysInMode is called again with an *iMode* value of 0 at the end of a recording session

The *iMode* and the *fhSysIn* values are obtained by this operation through Executive commands, such as **Record**, **Stop Record**, **Submit**, and **PlayBack**. (See the *CTOS Executive Reference Manual* for details on these commands.)

Procedural Interface

SetSysInMode (iMode, fhSysIn): ercType

where

iMode

is a word that is one of the following values:

Value	Mode
0	normal mode (neither recording mode nor playback mode is active)
1	recording mode (a copy of keyboard input is to be written to the recording file specified by *fhSysIn*)
2	playback mode (input is to be read from the submit file specified by *fhSysIn*)

fhSysIn

is the file handle (word) of the open file to use for the submit or recording file. The application program must make no further reference to this file. *fhSysIn* is not used if *iMode* is 0 (normal mode).

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	59
12	fhSysIn	2	
14	iMode	2	

This page intentionally left blank

SetTerminal has no procedural interface. You must make a request block and issue the request using Request, RequestDirect, or RequestRemote.

Caution: *This operation works on the shared resource processor CP and TP boards only. Do not use it on workstation hardware.*

Description

SetTerminal is used by the client to perform out-of-band functions on a port. Out-of-band functions include changing the baud rate of the port, turning on XON/XOFF recognition, sending break requests, and so forth.

Procedural Interface

SetTerminal has no procedural interface. You must make a request block and issue the request using one of the request operations listed above

where

bDestCPU

is the slot number (byte) of the Cluster or Terminal Processor connected to an RS-232-C port.

bSourceCpu

is the slot number (byte) of the client.

bPort

is the port number (byte) of a terminal attached to an RS-232-C port.

bWindow

is a byte value that must be 0.

bFunctionCode

is a byte with one of the following values:

Value	**Function**
0	Read parameter block. Reads the parameter block described below.
1	Write parameter block. Writes the parameter block.
2	Flush input buffer. Empties the input buffer.
3	Flush output buffer. Empties the output buffer.
4	Send break. Sends a 1/4-second break.
5	Suspend output. Stops output.
6	Resume output. Resumes output that has been suspended.

bDummy

is a byte value used for aligning the memory addresses on word boundaries.

The format of the parameter block is as follows:

Offset	**Field**	**Size (Bytes)**	**Description**
0	bBaud	1	A value from the baud rates below
1	bStopBits	1	0, 1, 2 but 0 = 1.5 stop bits
2	bParity	1	0, 1, 2 = odd, even, none
3	bCharBits	1	5, 6, 7, or 8 data bits
4	fFlowGen	1	If TRUE, then send XON/XOFF.
5	fFlowAct	1	If TRUE, then accept XON/XOFF sequences from the terminal.
6	fFlowAny	1	If TRUE, then any character will serve as an XON. This field is ignored unless fFlowAct is TRUE.
7	fLeadinEnable	1	If TRUE, then every <01> character in the input is followed by a byte that gives a delay time in 20-millisecond units.
8	fNoHangOnClose	1	If TRUE, then the line will not be placed on hook when a CloseTerminal request is received.
9	bOwner	1	Read only, slot number of port owner.

The *bBaud* field of the parameter block above uses the following baud rate values:

bBaud	**Rate**	**bBaud**	**Rate**	**bBaud**	**Rate**
1	50	6	200	11	2400
2	75	7	300	12	480
3	110	8	600	13	9600
4	134.5	9	1200	14	19200
5	150	10	1800		

pParamBlock
sParamBlock

describe the memory area of the parameter block to be written when SetTerminal is called with a *bFunctionCode* value that is not 0.

pParamRet
sParamRet

describe the memory area into which the parameter block is returned when SetTerminal is called with a *bFunctionCode* value of 0 (read).

The format for the parameter block, which is described above, is the same for *pParamBlock* and *pParamRet*.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	ExchResp	2	
8	ercRet	2	
10	rqCode	2	12310
12	bDestCPU	1	
13	bSourceCPU	1	
14	bPort	1	
15	bWindow	1	
16	bFunctionCode	1	
17	bDummy	1 (for alignment)	
18	pParamBlock	4	
22	sParamBlock	2	
24	pParamRet	4	
28	sParamRet	2	

This page intentionally left blank

SetTimerInt (pTpib): ercType

Description

The SetTimerInt primitive starts the Programmable Interval Timer (PIT). The caller provides the address of a timer pseudointerrupt block (TPIB) in local memory. When the specified time interval expires, SetTimerInt performs either of two functions depending upon the value in the *pIntHandler* field in the TPIB. It can

- Call a pseudointerrupt handler in the client to receive a pseudointerrupt. This option causes the client program to be locked into memory.
- Send a message to an exchange. The message is the memory address of the TPIB. This option does not cause the client program to be locked into memory.

SetTimerInt can be called from a PIT pseudointerrupt handler to reestablish the PIT pseudointerrupt handler with a (possibly different) interval. Multiple pseudointerrupt handlers can use the PIT simultaneously.

(See Chapter 4, "System Structures," for the format of a TPIB.)

NOTE: SetTimerInt guarantees a minumum time interval only.

Procedural Interface

SetTimerInt (pTpib): ercType

where

pTpib

is the memory address of the TPIB.

Request Block

SetTimerInt is a Kernel primitive.

SetTrapHandler (iTrap, pTrapHandler): ercType

Description

SetTrapHandler is used to set a raw interrupt handler to serve software generated interrupts (traps). Unlike SetIntHandler, SetTrapHandler does not disable swapping of the caller.

The handler specified by SetTrapHandler is part of the process context for all processes with the same user number. A process with a different user number can specify a different trap handler for the same interrupt level. Any process that has not specified a trap handler for a given interrupt level will receive the default system trap handler.

On protected mode systems prior to CTOS II 3.3, SetTrapHandler returns an error if the specified interrupt level is currently serving a hardware generated interrupt. On CTOS II 3.3 and higher, an application may specify a trap handler for any of the interrupt levels (0 to 255) except level 14 (Page Fault Exception) and level 205 (CTOS Request Interface).

Procedural Interface

SetTrapHandler (iTrap, pTrapHandler): ercType

where

iTrap

is a byte value indicating the trap type (0 to 255).

pTrapHandler

is the memory address of the trap handler entry point.

Request Block

fRaw is always 255.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	12
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	268
12	iTrap	2	
14	pTrapHandler	4	
18	reserved	4	
22	fRaw	2	255

SetUpMailNotification (pbMailUser, cbMailUser, pbMailCenter, cbMailCenter, pbMailPassword, cbMailPassword, bMinPrecedence): ercType

Description

SetUpMailNotification sets up the environment so that subsequent calls to QueryMail will correctly notify the user of incoming mail.

Signon.run uses this operation. It is only useful to programmers who intend to write their own replacement for Signon but still want users to receive mail notification.

Procedural Interface

SetUpMailNotification (pbMailUser, cbMailUser, pbMailCenter, cbMailCenter, pbMailPassword, cbMailPassword, bMinPrecedence): ercType

where

pbMailUser
cbMailUser

describe the mail login name. Signon looks for the name in the user file. The token is ":MailUserName:"

pbMailCenter
cbMailCenter

describe the mail center of the mail user. Signon looks for the center name in the user file. The token is ":MailCenterName:"

pbMailPassword
cbMailPassword

describe the mail login password. Signon looks for the password in the user file. The token is ":MailUserPassword:"

bMinPrecedence

is a byte that indicates the lowest precedence for which notification will take place. The allowed values are

Value	**Description**
0	Low
1	Normal
2	Urgent
255	None

Request Block

SetUpMailNotification is an object module procedure.

SetUserFileEntry (pNlsTableArea, pBuffer, sBuffer, pbFieldName, cbFieldName, pbValue, cbValue): ercType

Description

SetUserFileEntry modifies an operator's .user file by opening the file, searching for a ":FieldName:" string entry, updating the string value found following the trailing colon of the entry, and then closing the file. Unlike the SetField operation, which does not assume unique field names in the file and begins searching at the current file location, SetUserFileEntry assumes that each field name is unique and begins searching at the beginning of the file.

If the entry is not found, the ":FieldName:" string as well as the specified value are appended to the .user file.

If a Native Language Support (NLS) table area is present at system boot, the name of the .user file can be nationalized. See the description of *pNlsTableArea.* (For details on NLS, see "Native Language Support" in the *CTOS Operating System Concepts Manual.*)

Procedural Interface

SetUserFileEntry (pNlsTableArea, pBuffer, sBuffer, pbFieldName, cbFieldName, pbValue, cbValue): ercType

where

pNlsTableArea

The user file suffix .user is constructed differently based on the following:

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was not present at system boot, the suffix for the user file is .user.
- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was present at system boot, the suffix is constructed based on the value in the NLS Strings table.
- If *pNlsTableArea* is the memory address of an alternate set of NLS tables, the suffix is constructed based on the value in the NLS Strings table in the alternate set of NLS tables.

pBuffer
sBuffer

describe a buffer to be used in the search. *sBuffer* (word) must be a multiple of 512.

pbFieldName
cbFieldName

describe the field name entry to be searched for. The colons preceding and following the entry should not be given as part of the field name.

pbValue
cbValue

describe the string to be inserted following the field name trailing colon.

Request Block

SetUserFileEntry is an object module procedure.

This page intentionally left blank

SetVideoTimeout (nMinutes): ercType

NOTE: This operation is supported on protected mode operating systems only.

Description

SetVideoTimeout is used to have the screen refresh turned off after a specified time has elapsed during which no keyboard activity has occurred.

Procedural Interface

SetVideoTimeout (nMinutes): ercType

where

nMinutes

is the number of minutes (word) in the range of 0 to 109 until the timeout occurs. A value of 0 means that no timeout occurs.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	255
12	nMinutes	2	

SetWsUserName (pbUserName, cbUserName): ercType

Description

SetWsUserName stores the user number SignOn name of the workstation. SetWsUserName is used primarily by the SignOn program, which also places the user name in the Application System Control Block of the server workstation in a cluster configuration. (See the *CTOS Programming Guide* for more information.)

Procedural Interface

SetWsUserName (pbUserName, cbUserName): ercType

where

pbUsername
cbUserName

describe the user number sign-on name to be stored.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
1	RtCode	1	
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	203
12	reserved	6	
18	pbUserName	4	
22	cbUserName	2	

SetXBusMIsr (pIhRet, pIntHandler, saData): ercType

Caution: *This operation works on workstation hardware only. Do not use it on shared resource processor hardware.*

Description

SetXBusMIsr establishes an XINT4 multiplexed interrupt service routine (ISR). It also controls the allocation of dedicated XINT0 or XINT1 ISRs.

(See the *CTOS Operating System Concepts Manual* for more information on X-Bus interrupts.)

Procedural Interface

SetXBusMIsr (pIhRet, pIntHandler, saData): ercType

where

pIhRet

is the memory address of the word to which the interrupt handle is returned if you are using SetXBusMIsr to establish XINT4.

If you are using SetXBusMIsr to allocate the XINT0 or XINT1 ISR (*pIntHandler*=0), *ihRet* tells you which ISR was allocated for the interrupt. *ihRet* contains the index of the hardware interrupt vector corresponding to the interrupt. This index must be passed to SetIntHandler to establish the interrupt handler.

pIntHandler

is the memory address of the interrupt handler entry point. The interrupt handler is of type boolean.

If *pIntHandler*=0, XINT0 or XINT1 is being allocated rather than XINT4. Note that if *pIntHandler*=0, a subsequent SetIntHandler call must be made to establish the handler.

saData

is word value that is either the segment base address (sa) of the data segment that is to be used by the XINT4 mediated interrupt handler, or a bit mask indicating that XINT0 and/or XINT1 are to be allocated.

The bit mask interpretations are

Bit	**ISR Allocated**
1	XINT0
2	XINT1
3	XINT0 and XINT1

Request Block

sIhRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32799*
12	pIntHandler	4	
16	saData	2	
18	pIhRet	4	
22	sIhRet	2	2

*On BTOS ii, 3.2 workstation operating systems, the request code 8204 is also supported.

This page intentionally left blank

SgFromSa (sa, userNumSaOwner, pSgRet): ercType

Caution: *This operation should be used only by programs executing in protected mode.*

Description

SgFromSa returns an alias Global Descriptor Table (GDT) selector (SG) that references the same memory location as the specified segment address (SA).

The SA can be

- a GDT selector (SG)
- a Local Descriptor Table (LDT) selector (SL)
- a real mode segment address (SN)

Caution: *If the SA is an SN, the SN should be an equivalent pointer with the lowest possible offset. Otherwise, status code 407 ("Invalid pointer") may be returned.*

SgFromSa is used, for example, by VAM to reference a memory location in an application program executing in the following modes:

- real or protected, on an operating system executing in protected mode
- real mode, on a real mode operating system

The application program mode can be determined by *userNumSaOwner*.

Procedural Interface

SgFromSa(sa, userNumSaOwner, pSgRet): ercType

where

sa

is a word value that is a Global Descriptor Table (GDT) selector (SG), a Local Descriptor Table (LDT) selector (SL), or a real mode segment address (SR) to which an SG alias is to be created.

userNumSaOwner

is the user number (word) returned from a CreatePartition or a GetPartitionHandle operation. If *userNum* is 0, then the user number of the calling program is used.

pSgRet

is the memory address of a word to which the alias SG selector is to be returned.

Request Block

sSgMax is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	332
12	reserved	2	
14	userNumSaOwner	2	
16	sa	2	
18	pSgRet	4	
22	sSgMax	2	2

This page intentionally left blank

ShortDelay (n): ercType

Description

ShortDelay suspends the execution of the client for the specified interval.

ShortDelay is similar to the Delay operation except that, with ShortDelay, the unit interval is 1 millisecond rather than 100 milliseconds. Because of the overhead in setting up the timer, the shorter the interval to delay, the less accurate ShortDelay becomes. Unlike Delay or Doze, ShortDelay uses the Programmable Interval Timer (PIT), not the Realtime Clock.

ShortDelay suspends client execution for a maximum of 3276 milliseconds. If periods longer than 100 milliseconds are required, Delay or Doze should be used.

Procedural Interface

ShortDelay (n): ercType

where

n

is a word value that indicates the interval to delay (in units of 1 millisecond).

Request Block

ShortDelay is an object module procedure.

This page intentionally left blank.

ShortDelay (n): ercType

Description

ShortDelay suspends the execution of the client for the specified interval.

ShortDelay is similar to the Delay operation except that, with ShortDelay, the unit interval is 1 millisecond rather than 100 milliseconds. Because of the overhead in setting up the timer, the shorter the interval to delay, the less accurate ShortDelay becomes. Unlike Delay or Doze, ShortDelay uses the Programmable Interval Timer (PIT), not the Realtime Clock.

ShortDelay suspends client execution for a maximum of 3276 milliseconds. If longer periods are required, Delay or Doze should be used.

Procedural Interface

ShortDelay (n): ercType

where

n

is a word value that indicates the interval to delay (in units of 1 millisecond).

Request Block

ShortDelay is an object module procedure.

This page intentionally left blank

ShrinkAreaLL (p, cBytes): ercType

Description

ShrinkAreaLL deallocates memory of the specified size within the specified long-lived segment. Memory must be deallocated in a sequence exactly opposite the one in which it was allocated (that is, last allocated, first deallocated).

ShrinkAreaLL differs from DeallocMemoryLL in that the offset portion of the memory pointer may be non-zero.

Use ShrinkAreaLL for a program designed to execute in both real and protected modes if the program will deallocate memory in a long-lived segment.

Procedural Interface

ShrinkAreaLL (p, cBytes): ercType

where

p

is the memory address of the memory to be deallocated.

cBytes

is the count (word) of bytes of memory to be deallocated.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	282
12	cBytes	2	
14	p	4	

ShrinkAreaSL (p, cBytes): ercType

Description

ShrinkAreaSL deallocates memory of the specified size within the specified short-lived segment. The segment must have been created by a prior call to AllocAreaSL (except when using the Linker's DS Allocation facility and ShrinkAreaSL to shrink the static data segment, DGroup). Memory must be deallocated in a sequence exactly opposite the one in which it was allocated (that is, last allocated, first deallocated).

ShrinkAreaSL differs from DeallocMemorySL in that the offset portion of the memory address may be non-zero.

Use ShrinkAreaSL for a program designed to execute in both real and protected modes if the program will deallocate memory in a short-lived segment.

Procedural Interface

ShrinkAreaSL (p, cBytes): ercType

where

p

is the memory address of the memory to be deallocated.

cBytes

is the count (word) of bytes of memory to be deallocated.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	281
12	cBytes	2	
14	p	4	

ShrinkPartition: ercType

Description

ShrinkPartition reduces the size of the application partion to the amount of memory currently allocated to short–lived memory. ShrinkPartition succeeds if and only if all long-lived memory has been previously deallocated. Use the ResetMemoryLL operation to deallocate long-lived memory.

ShrinkPartition typically is used by a program installing itself as a system service. Before calling ConvertToSys, ShrinkPartition is called to make more memory available to other partitions.

Procedural Interface

ShrinkPartition: ercType

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	0
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	341

This page intentionally left blank

SignOnLog has no procedural interface. You must build a request block and issue the request using Request, RequestDirect, or RequestRemote.

Description

SignOnLog is issued by the SignOn program after validation of access to and contents of the user file and before user access to the system is allowed. If the request is served by a system service and is returned with an error served (that is, SignOn gets back a status code other than 0, 6, 31 or 33), access is not allowed.

Procedural Interface

SignOnLog does not have a procedural interface. You must build a request block and issue the request using one of the request Kernel primitives listed above

where

pbUserName
cbUserName

describe the SignOn user name.

pbPassword
cbPassword

describe the user password, if any.

seconds

is a word value indicating the count (0–43199) of seconds since the last midnight/noon.

dayTimes2

is a word value indicating the count (0–65535) of 12 hour periods since March 1, 1952.

pSignonLogData

is reserved for future use as the memory address of client data.

sSignonLogData

is reserved for future to indicate the client data size.

pSignonLogDataRet

is reserved for future use as the memory address of a buffer into which the data is to be returned to a client from a system service.

sSignonLogDataRetMax

is reserved for future use to indicate the maximum buffer size.

psSignonLogDataRet

is reserved for future use as the memory address of a buffer into which the count of bytes in the data returned is to be placed.

Request Block

ssSignonLogDataRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	3
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	4155
12	seconds	2	
14	dayTimes2	2	
16	reserved	2	
18	pbUserName	4	
22	cbUserName	2	
24	pbPassword	4	
28	cbPassword	2	
30	pSignonLogData	4	
34	sSignonLogData	2	
36	pSignonLogDataRet	4	
40	sSignonLogDataRetMax	2	
42	psSignonLogDataRet	4	
46	ssSignonLogDataRet	2	2

This page intentionally left blank

SnFromSr (sr, pSnRet): ercType

Caution: *This operation is supported only by operating systems executing in protected mode.*

Description

SnFromSr returns the protected mode selector (SN) corresponding to the real mode segment address (SR).

SnFromSr supports protected mode system–common procedures that need to access memory using real mode segment addresses. SnFromSr succeeds only if the system–common procedure caller is executing in real mode.

For example, the caller could be the OFIS Document Designer executing in real mode on an operating system executing in protected mode. If the OFIS Document Designer calls certain Video Access Method (VAM) system–common procedures, VAM must call SnFromSr to translate the OFIS Document Designer's real mode segment address into its corresponding protected mode selector.

Procedural Interface

SnFromSr (sr, pSnRet): ercType

where

sr

is the real mode segment address (word).

pSnRet

is the memory address to which the protected mode selector is returned.

Request Block

SnFromSr is a system–common procedure.

SpoolerPassword (pbPrinterName, cbPrinterName, pbPassword, cbPassword): ercType

Description

SpoolerPassword sends a file password to the spooler. If the spooler is in the security mode, it waits for the password before it proceeds to open and read the protected file.

Procedural Interface

SpoolerPassword (pbPrinterName, cbPrinterName, pbPassword, cbPassword): ercType

where

pbPrinterName
cbPrinterName

describe the name of the printer.

pbPassword
cbPassword

describe the password.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	189
12	reserved	6	
18	pbPrinterName	4	
22	cbPrinterName	2	
24	pbPassword	4	
28	cbPassword	2	

SpoolerVersion (pVerStruct, sVerStruct, pcbRet): ercType

Description

SpoolerVersion returns a structure that identifies the version of the Spooler, indicates whether to print initial form feed characters, and specifies the name of the spooler configuration file.

Procedural Interface

SpoolerVersion (pVerStruct, sVerStruct, pcbRet): ercType

where

pVerStruct
sVerStruct

describe the memory area to which the structure containing the version and installation parameters is written. The size of the memory area should be 4 bytes plus the number of bytes in the Spooler configuration file specification. If insufficient memory is provided, the returned data will be truncated. The format of the structure is as follows:

Offset	**Field**	**Size (Bytes)**
0	version	2
2	fPrint1stFF	1
3	cbCnfgFileName	1
4	rgbCnfgFileName	varies

pcbRet

describes a word into which the actual length of the returned structure is placed.

Request Block

scbRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8211
12	reserved	6	
18	pVerStruct	4	
22	sVerStruct	2	
24	pcbRet	4	
28	scbRet	2	2

SrFromSn (sn, userNum, pSrRet): ercType

Caution: *This operation is supported only by operating systems executing in protected mode.*

Description

SrFromSn returns the real mode segment address (SR) corresponding to the protected mode selector (SN).

SrFromSn supports protected mode system services that maintain system data structures to be accessed by real mode application programs. For example, system services have to return real mode memory addresses to their real mode clients.

Procedural Interface

SrFromSn (sn, userNum, pSrRet): ercType

where

sn

is the protected mode selector (word).

userNum

is the user number (word) returned from a CreatePartition or a GetPartitionHandle operation. If *userNum* is 0, then the user number of the calling program is used.

pSrRet

is the memory address to which the real mode segment address is returned.

Request Block

SrFromSn is a system-common procedure.

StatisticsVersion (pVerStruct, sVerStruct, pcbRet): ercType

Description

StatisticsVersion returns a structure that contains version of the installed Performance Statistics System Service.

Procedural Interface

StatisticsVersion (pVerStruct, sVerStruct, pcbRet): ercType

where

pVerStruct
sVerStruct

describe a memory area to which the structure containing the version is written. The format of the structure is as follows:

Offset	**Field**	**Size (Bytes)**
0	version	2

pcbRet

describes a word into which the length of the returned structure is placed.

Request Block

scbRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	1	
6	exchResp	1	
8	ercRet	1	
10	rqCode	1	8230
12	reserved	6	
18	pVerStruct	4	
22	sVerStruct	2	
24	pcbRet	4	
28	scbRet	2	2

StringsEqual (psbString1, psbString2): Boolean

NOTE: For ease in nationalization, new programs should use NlsULCmpB rather than this operation. (See NlsULCmpB.) StringsEqual is documented for historical reasons only.

Description

StringsEqual is a Boolean function that returns TRUE if the two strings are the same. Uppercase and lowercase are ignored. For example, if the "Exec.Run" and "exec.run" strings are compared, the function returns TRUE.

Procedural Interface

StringsEqual (psbString1, psbString2): Boolean

where

psbString1
psbString2

are the 4-byte memory addresses of the two strings to be compared, where the first byte of each string is its size.

Request Block

StringsEqual is a system-common procedure.

This page intentionally left blank

SuspendProcess (pid): ercType

Description

SuspendProcess suspends the specified process. The identification number (ID) of the process must be a valid number such as that returned by QueryProcessNumber. A suspension asserted using SuspendProcess may be deasserted by calling the UnsuspendProcess operation.

Procedural Interface

SuspendProcess (pid): ercType

where

pid

specifies the ID of the process to be suspended.

Request Block

SuspendProcess is a Kernel primitive.

This page intentionally left blank

SuspendUser (userNum): ercType

Description

SuspendUser suspends all processes of the specified user number (partition). Specifying a 0 value for *userNum* suspends all processes in the caller's partition. A suspension asserted using SuspendUser may be deasserted by calling the UnsuspendUser operation.

Procedural Interface

SuspendUser (userNum): ercType

where

userNum

specifies the user number (word) whose processes are to be suspended.

Request Block

SuspendUser is a Kernel primitive.

This page intentionally left blank

SwapContextUser(userNum): ercType

Description

SwapContextUser is used by an asynchronous system service to handle swapping requests. The operation works very much like TerminateContextUser. (See the description of TerminateContextUser.) With SwapContextUser, however, the system service is not finished with the client request. When the user is swapped back into memory, the context must resume execution where it left off.

Procedural Interface

SwapContextUser(userNum): ercType

where

userNum

is the user number (word) associated with the terminating program.

Request Block

SwapContextUser is an object module procedure in Async.lib.

This page intentionally left blank

SwapInContext (userNum): ercType

Description

SwapInContext asks the operating system to swap a specified user number from disk into memory.

SwapInContext returns ercOK after the user number is swapped into memory. If the user number is already in memory, ercOK is returned immediately.

An error may be returned if the specified user number cannot be swapped in because of fragmentation caused by semi-permanently locked memory. SwapInContext will, however, wait for temporarily locked memory, such as that used by an input/output function, to become available.

Procedural Interface

SwapInContext (userNum): ercType

where

userNum

is the user number (word) to be swapped in.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	295
12	userNum	2	

Swapping requests have no procedural interface.

NOTE: This page presents the standard format for all swapping requests.

Description

Swapping requests are issued to all system services by the operating system when the operating system suspends and swaps a program to disk. They ensure that no responses are made to any program that is not resident in memory.

When a system service receives a swapping request, it is required to respond to all outstanding requests with the same client user number and then to respond to the swapping request.

The system service can use either of the following strategies:

- It can hold the swapping request until it completes the processing of all outstanding requests for the client. It must respond immediately to each outstanding request and then respond normally to the swapping request.
- It can respond to all outstanding requests for the client with status code 37 ("Service not completed"). The operating system intercepts this special response status code, and the program is swapped to disk. Later, when the program is swapped back into memory, the operating system reissues the original outstanding requests to the system service.

A program is totally unaware that it is being swapped out of memory or that any of its requests are being handled other than in the usual manner.

A filter that uses only the replacement method should have its own swapping request. In this case it is the same as a normal system service described above. (Filter types are described in "System Services Management" in the *CTOS Operating System Concepts Manual*.)

A filter that uses one of the pass-through methods must filter the swapping requests of the original system service(s). If the filter uses only two-way pass-through method for any requests, the filter must use that method for the swapping requests.

(See Appendix D for a list of the operating system swapping requests and request codes that are issued.)

If optional system services are installed, which define other swapping requests, the operating system issues these requests along with those listed in Appendix D.

Procedural Interface

Swapping requests have no procedural interface.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	*
12	reserved	2	

*Request code of swapping request

SwapXBusEar (newEar): WORD

Caution: *Do not use this operation on shared resource processor or SuperGen Series 5000 hardware.*

Description

The word value returned by SwapXBusEar is the value that was previously written to the register by the Kernel. Any operations using this operation are responsible for restoring the previous value when they have finished accessing X-Bus memory. Interrupt service routines are the only operations that should need to use SwapXBusEar, since the Kernel sets up the Extended Address Register (EAR) automatically for operations executing at process level.

Procedural Interface

SwapXBusEar (newEar): WORD

where

newEar

is the new value to be written to the X-Bus EAR.

Request Block

SwapXBusEar is a system-common procedure.

This page intentionally left blank

SystemCommonCheck (iSystemCommon): ercType

Caution: *This operation is provided for backward compatibility only and, therefore, should not be used. Use SystemCommonQuery instead.*

Description

SytsemCommonCheck is used to determine if a system–common procedure has been served.

The procedure returns status code 7 ("Not implemented") if the system–common procedure was not served.

Procedural Interface

SystemCommonCheck (iSystemCommon): ercType

where

iSystemCommon

is the number of the system–common procedure to be checked.

Request Block

SystemCommonCheck is a system–common procedure.

This page intentionally left blank

SystemCommonConnect (iSystemCommon, pProcedure): ercType

Caution: *This operation is provided for backward compatibility only and, therefore, should not be used. Use SystemCommonInstall instead.*

Description

SystemCommonConnect causes a procedure to be installed for the given system–common procedure number (*iSystemCommon*).

Status code 5 ("Syntax error") is returned if no parameter list exists for the procedure intended to be installed. A parameter list would not exist if the following are TRUE:

- The list was not defined in [Sys]<Sys>Config.Sys.
- The list was not built into the operating system by means of a SysGen. (Parameter lists for the video access method (VAM) operations, for example, are built–in lists.)

Status code 8 ("Internal inconsistency") is returned if the procedure specified by *iSystemCommon* already is installed.

Procedural Interface

SystemCommonConnect (iSystemCommon, pProcedure): ercType

where

iSystemCommon

is the number of the system–common procedure to install.

pProcedure

is the memory address of the system-common procedure. This address must be GDT-based. To ensure that it is, enter "GDTProtected" in the RunFileType field of the Bind command when linking the program that is to call SystemCommonConnect.

Request Block

SystemCommonConnect is a system-common procedure.

SystemCommonInstall (iSystemCommon, pProcedure, pbParamDef, cbParamDef, tyOperation, pInfoRet, sInfoRet): ercType

Caution: *This operation should be used only by programs executing in protected mode.*

Description

SystemCommonInstall installs *pProcedure*, the address of a system-common procedure. The address, *pProcedure*, must be GDT-based.

The character string that defines the parameters for the procedure being installed can subsequently be used by the operating system for creating alias pointers. For example, when the operating system intercepts (traps) calls made by real mode programs to the system-common procedure, it creates alias GDT selectors for each pointer it identifies in the character string. (See Appendix I for the character string syntax used to define parameters.)

Procedural Interface

SystemCommonInstall (iSystemCommon, pProcedure, pbParamDef, cbParamDef, tyOperation, pInfoRet, sInfoRet): ercType

where

iSystemCommon

is the number of the system-common procedure.

pProcedure

is the memory address of the system-common procedure serving the system-common entry. To ensure that this address is GDT-based, enter "GDTProtected" in the RunFileType field of the **Bind** command when linking the system-common service that will call SystemCommonInstall.

pbParamDef
cbParamDef

describe the memory area that defines the number of parameters required by the procedure being installed, and their types. (See Appendix I for the character string syntax used to define parameters.)

tyOperation

is a word that describes the type of operation desired.

If *tyOperation*=1, it is legal to replace the currently installed procedure, if one exists, with the procedure referenced by *pProcedure*. If *tyOperation*=0, this operation will fail if a procedure is already installed. Be certain to push a full word onto the stack for this parameter and do not pass any values other than 0 and 1. Future operating systems may interpret additional values.

pInfoRet
sInfoRet

describe the memory area to which the information about an existing system–common procedure (a procedure being replaced by the installed procedure) is to be copied. This information can be used, for example, by a filter program to filter existing system–common calls. The first *sInfoRet* bytes of the following data structure will be copied there:

Offset	Field	Size (Bytes)
0	fDefined	1
1	pProc	4
5	orgbParamDef	2
7	srgbParamDef	2
	orgbParamDef	
	rgbparamDef	
	srgbParamDef	

The field, *orgbParamDef*, is an offset within this data structure pointing to the parameter definition string (*rgbParamDef*). (See Appendix I for the format of the parameter definitions in *rgbParamDef*.) Your program should not inspect this returned string, because it may change in future operating system releases. Any particular operating system actually uses only a subset of the options described in Appendix I. Thus, the string returned will reflect only those options used by the currently running operating system, which is sufficient for subsequent calls to SystemCommonInstall.

If *fDefined* is FALSE, the system–common procedure is not defined and the remaining fields should not be referenced.

Request Block

SystemCommonInstall is a system–common procedure.

This page intentionally left blank

SystemCommonQuery (iSystemCommon, pInfoRet, sInfoRet): ercType

Description

SystemCommonQuery returns information about the requested system-common entry.

This operation can be used to determine if the specified system-common procedure is defined.

Procedural Interface

SystemCommonQuery (iSystemCommon, pInfoRet, sInfoRet): ercType

where

iSystemCommon

is the number of the system-common procedure to be queried.

pInfoRet sInfoRet

describe the memory area to which the information about the system–common procedure is to be copied. The first *sInfoRet* bytes of the following data structure will be copied there:

Offset	**Field**	**Size (Bytes)**
0	fDefined	1
1	pProc	4
5	orgbParamDef	2
7	srgbParamDef	2
orgbParamDef	rgbparamDef	srgbParamDef

The field, *orgbParamDef*, is an offset within this data structure pointing to the parameter definition string (*rgbParamDef*). (See Appendix I for the format of the parameter definitions in *rgbParamDef*.) Your program should not inspect this returned string, because it may change in future operating system releases. Any particular operating system actually uses only a subset of the options described in Appendix I. Thus, the string returned will reflect only those options used by the currently running operating system, which is sufficient for subsequent calls to SystemCommonInstall.

If *fDefined* is FALSE, the system–common procedure is not defined and the remaining fields should not be referenced.

Since all system services run in protected mode, the memory address of the procedure (*pProc*) returned by this operation will be GDT–based. Real mode programs are unable to use such addresses, and therefore can call this operation only if *sInfoRet*=1.

Request Block

SystemCommonQuery is a system–common procedure.

TerminateAllOtherContexts(erc): ercType

Description

TerminateAllOtherContexts is called by an asynchronous system service at deinstallation to terminate each active context.

Procedural Interface

TerminateAllOtherContexts(erc): ercType

where

erc

is the status code that is responded to for any outstanding requests associated with the context being terminated.

Request Block

TerminateAllOtherContexts is an object module procedure in Async.lib.

This page intentionally left blank

TerminateContext: ercType

Description

TerminateContext is used by an asynchronous system service to terminate a context and return its stack space to the heap.

Like ResumeContext, TerminateContext does not take any arguments. It uses a global offset that points to the current Context Control Block (CCB) in the heap. Any heap space associated with the context is returned to the heap at this time. After terminating the context and returning the heap memory, TerminateContext returns to the caller's wait loop. A status code is returned only if the offset to the current CCB does not point to a valid context.

Procedural Interface

TerminateContext: ercType

Request Block

TerminateContext is an object module procedure in Async.lib.

This page intentionally left blank

TerminateContextUser (userNum, erc): ercType

Description

TerminateContextUser is used by an asynchronous system service to respond to all client requests for a given user number.

Procedural Interface

TerminateContextUser (userNum, erc): ercType

where

userNum

is the user number associated with the terminating program. The user number is obtained from the termination request.

erc

is the status code number.

Request Block

TerminateContextUser is an object module procedure in Async.lib.

This page intentionally left blank

TerminatePartitionTasks (userNumPartition): ercType

Description

TerminatePartitionTasks terminates the program in the application partition specified by the user number and loads and activates the partition's exit run file.

If the partition is locked, status code 806 ("Partition is locked") is returned and a flag is set in the Application System Control Block to notify the program in the partition.

Procedural Interface

TerminatePartitionTasks (userNumPartition): ercType

where

userNumPartition

is the user number (word) returned from a CreatePartition or a GetPartitionHandle operation.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	179
12	userNumPartition	2	

TerminateQueueServer (pbQueueName, cbQueueName): ercType

Description

TerminateQueueServer is used by the queue server to notify the Queue Manager that it is no longer serving the specified queue. TerminateQueueServer should be called when termination is the result of normal circumstances.

TerminateQueueServer unmarks any queue entries that were marked by the terminating queue server.

Procedural Interface

TerminateQueueServer (pbQueueName, cbQueueName): ercType

where

pbQueueName
cbQueueName

describe a queue name corresponding to a queue name specified when the queue was installed.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	0
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	147
12	pbQueueName	4	
16	cbQueueName	2	

Termination requests have no procedural interface.

NOTE: This page presents the standard format for all termination requests.

Description

Termination requests function similarly to abort requests in that they notify all system services that a client program has terminated. Upon notification, the system services can release resources, such as open files and locked ISAM records, that they have allocated to the terminating program.

The operating system issues termination requests when the program

- terminates by calling Chain, Exit, ErrorExit, or ErrorExitString
- is terminated by ACTION-FINISH or by TerminatePartitionTasks

(See Appendix D for a list of the operating system termination requests and request codes that are issued.)

If optional system services are installed that define other termination requests, the operating system issues these requests along with those listed in Appendix D.

In addition to termination requests, the operating system issues abort requests at a server workstation when the server detects that it cannot communicate with a cluster workstation or when a partition is vacated with the VacatePartition operation or through lack of an exit run file. (See the AbortRequests operation page in this chapter for more information.)

Procedural Interface

Termination requests have no procedural interface.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	*
12	ercTermination	2	**

* Request code of termination request

**0, if Exit
4, if ACTION-FINISH
erc passed to ErrorExit, if Chain, ErrorExit, or ErrorExitString

TextEdit (ch, pTdb): ercType

Description

TextEdit edits a line of text. The operation takes a character and a text descriptor and returns the descriptor with appropriate changes.

TextEdit does not modify the screen or place text on it. That is done by the application program, using PutFrameChars, PutFrameAttrs, and PosFrameCursor.

TextEdit is called by the SignOn program, for example, to allow a user to edit the SignOn form.

Status code 2500 ("Buffer full") is returned if the character buffer becomes full. Status code 2501 ("Invalid character") is returned if a character is not one of those mentioned below.

Procedural Interface

TextEdit (ch, pTdb): ercType

where

ch

is a character.

The following are treated as normal ASCII characters:

03h, 06h, and 10h
20h to 5Bh
5Dh to 7Eh
0A0h to 0B5h

These control characters mean normal character input, inserting (if overtype LED is off) or overwriting at the cursor location, and then incrementing the cursor location.

Control Character	**Description**
0	Basic Conversation
0A7h	(code single quote) means treat the next character as normal character input. This value can be configured using the Native Language Support (NLS) Special Characters table. (For details on NLS, see the *CTOS Operating System Concepts Manual.)*
0D0h	(overtype) means toggle the overtype LED.
012h	(right arrow) means move the cursor to the right one position.
0D2h	(shift right arrow) means move the cursor to the right five positions.
092h	(code right arrow) means move the cursor to the end of text.
00Eh	(left arrow) means move the cursor to the left one position.
0CEh	(shift left arrow) means move the cursor to the left five positions.
08Eh	(code left arrow) means move the cursor to the beginning of text.
07Fh	(delete) means delete the character at the cursor position.
0C8h	(shift delete) means delete all characters starting at the cursor position.
0FFh	(code delete) means delete all characters.
008h	(backspace) means move the cursor left one space and delete the character there if overtype LED is off.

pTdb

is the memory address of the Text Descriptor Block, which has the following format:

Offset	Field	Size (Bytes)	Description
0	prgch	4	memory address of an array of characters. All characters in this character buffer should be set to 0 before TextEdit is called the first time.
4	cchMax	2	maximum number of characters that can be placed in the buffer (the size of the buffer).
6	cchMac	2	current number of characters in the buffer.
8	ichCursor	2	position of the cursor in the buffer.

Request Block

TextEdit is an object module procedure.

This page intentionally left blank

TransmitCommLineDma (commlinehandle, pb, cb): ercType

Description

TransmitCommLineDma sets up the DMA controller to transfer a specified number of bytes from the specified memory buffer to the communications channel.

This operation returns immediately after the DMA controller has been set up rather than waiting for the specified number of bytes to be transmitted. An application may use the GetCommLineDmaStatus operation to query the number of bytes actually transmitted.

Because TransmitCommLineDma is a system-common procedure, it may be called from a communications interrupt handler.

Procedural Interface

TransmitCommLineDma (commlinehandle, pb, cb): ercType

where

commLineHandle

is a handle (word) returned by InitCommLine.

pb
cb

describe the data to be transferred to the communications channel.

Request Block

TransmitCommLineDma is a system-common procedure.

This page intentionally left blank

TruncateDaFile (pDawa, qiRecord): ercType

Description

TruncateDaFile truncates the open DAM file (that is, it removes all records beyond a specified point). All records having record numbers greater than the *qiRecord* parameter are deleted. If *qiRecord* is 0, all records in the DAM file are deleted.

TruncateDaFile can be called only if the file was opened in modify mode. If the file was opened in read mode, status code 209 ("File is read only") is returned.

Procedural Interface

TruncateDaFile (pDawa, qiRecord): ercType

where

pDawa

is the memory address of the same Direct Access Work Area that was supplied to OpenDaFile.

qiRecord

is a 32-bit unsigned integer specifying a record number. All records having record numbers greater than *qiRecord* are deleted.

Request Block

TruncateDaFile is an object module procedure.

This page intentionally left blank

TsConnect (iVpModule, th, device, fAdd): ercType

Description

TsConnect connects or disconnects the telephone unit and telephone lines.

Connecting a line that is on hold (see TsHold) terminates the hold condition.

Status code 11202 ("Invalid connection") will be returned if the connection is not allowed (such as connecting the telephone unit to a modem) or if it would require use of hardware circuits already connected.

Status code 11205 ("Bad handle") will be returned if the handle specified is not valid.

Procedural Interface

TsConnect (iVpModule, th, device, fAdd): ercType

where

iVpModule

is the Voice Processor module on the X-Bus (word), 1 being the first Voice Processor module to the right of the CPU module.

th

is a telephone handle (word) returned by TsOffHook.

device

specifies the device (word) to connect. The word values of *device* are described below:

Value	**Description**
0	telephone line 1
1	telephone line 2
2	telephone unit
3	telephone unit connected in monitor mode

fAdd

is a flag. If TRUE the device is added to the connection; otherwise it is removed.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	8
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32814
12	iVpModule	2	
14	th	2	
16	device	2	
18	fAdd	2	

This page intentionally left blank

TsDataChangeParams (lh, pbLineControl, cbLineControl): ercType

Description

TsDataChangeParams allows a program to change the parity, line control, and other parameters for an open data line.

The baud rate may not be changed after the line has been opened.

Procedural Interface

TsDataChangeParams (lh, pbLineControl, cbLineControl): ercType

where

lh

is the line handle (word) returned by an TsDataOpenLine call.

pbLineControl
cbLineControl

describe a communications data control structure. (For the format of this structure, see "Data Control Structure" in Chapter 4, "System Structures.")

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	14
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32810
12	lh	2	
14	pbLineControl	4	
18	cbLineControl	2	

TsDataCheckpoint (lh, cTimeout): ercType

Description

TsDataCheckpoint causes a data line to be checkpointed. The request returns only after all TsDataWrite requests have been returned and after the modem has transmitted the last character.

In the case of a timeout, any TsDataWrite requests will be returned with status code 11206 ("Telephone Service timeout").

Procedural Interface

TsDataCheckpoint (lh, cTimeout): ercType

where

lh

is the line handle (word) returned by an TsDataOpenLine call.

cTimeout

is the maximum amount of time (word) in units of 100ms that the caller wants to wait for any outstanding TsDataRead or TsDataWrite operations to complete (0FFFFh implies wait forever).

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	14
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32808
12	lh	2	
14	cTimeout	2	

TsDataCloseLine (lh, cTimeout): ercType

Description

TsDataCloseLine causes a data line to be closed and the call to be terminated. It may be used either to terminate a call and hang up the line or to convert a data call to a voice call. In the latter case it would be followed by a call to TsConnect to attach the telephone unit.

Procedural Interface

TsDataCloseLine (lh, cTimeout): ercType

where

lh

is the line handle (word) returned by an TsDataOpenLine call.

cTimeout

is the maximum amount of time (word) in units of 100ms that the caller wants to wait for any outstanding TsDataRead or TsDataWrite operations to complete (0FFFFh implies wait forever).

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	14
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32809
12	lh	2	
14	cTimeout	2	

TsDataOpenLine (pLhRet, pbLineSpec, cbLineSpec, pbLineControl, cbLineControl, pbDialString, cbDialString, pcbStringRet): ercType

Description

TsDataOpenLine starts a data session using the modem.

Procedural Interface

TsDataOpenLine (pLhRet, pbLineSpec, cbLineSpec, pbLineControl, cbLineControl, pbDialString, cbDialString, pcbStringRet): ercType

where

pLhRet

is the memory address of the line handle that will be returned when a call is accepted.

pbLineSpec
cbLineSpec

describe a phone line specification. The specification is composed of the device, the module, and the line. Each part is described below:

Specification part	Description
device	The device (optional) is of the form "[PHONE]", "[!phone]" or "{nyc}[phone]".
module	The module (optional) is a single digit (1 to 9), specifying which Voice Processor module is desired.
line	The line (required) is "1" or "A" for line 1, and "2" or "B" for line 2. Sample specifications are "[PHONE]1A" (module 1, line 1 at local node) "{Nyc}[Phone]2" (module 1, line 2 at node "Nyc") "[phone]41" (module 4, line 1 at local node)

pbLineControl
cbLineControl

describe a Data Control Structure. (For the format of this structure, see "Data Control Structure" in Chapter 4, "System Structures.")

pbDialString
cbDialString

describe the number to be dialed. The string contains characters as defined below:

Characters	**Function**
0-9,A-Y,a-y,*,#	are the dial or dual-tone multifrequency (DTMF) characters normally found on the touch pad or rotary switch of a standard telephone unit. *Q*, *q* ,*Z*, and *z* are not mapped.
	!A,!B,!C,!D are the DTMF characters A-D, which do not normally appear on the touch pad.
%	Switch to pulse dialing.
&	Switch to tone dialing.
@	Flash for the default flash time.
!@x	Flash for *x* units of 100ms (*x* is a byte).
-	Wait for dial tone.
!-	Wait for any tone.
?	Wait for the phone being called to answer.
~	Pause for the default pause time.
!~x	Pause for x units of 100ms (x is a byte).

Characters	**Function**
$tl	Generate part of a DTMF tone *t* for time *l* (units of 10ms). (*t* and *l* are byte values.) The primary use of this is to generate a single tone (for use in answering machines, for example). The tone generated is derived from a combination of the two frequencies specified by the upper (column) and lower (row) 4 bit nibbles of *t*. The valid values of the nibbles are 0Eh, 0Dh, 0Bh, and 07h. If both the upper and lower nibbles are valid, the tone generated will be one of the 16 DTMF tones as specified in the following table:

	0E0h	0D0h	0B0h	070h
0Eh	1	2	3	A
0Dh	4	5	6	B
0Bh	7	8	9	C
07h	*	0	#	D

For example, a value of 0E7h will generate the tone for "*".

If only one of the nibbles is valid (such as 07h or 0E0h) a single tone will be generated. If neither nibble is valid no tone is generated.

SP,(,),-,/	These characters are ignored (*SP* is a space, or ASCII 20h).

pcbStringRet

is the memory address of a return status structure that contains the number of dial characters processed.

Request Block

sLhRet and *scbStringRet* are always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	0
1	RtCode	1	65
2	nReqPbCb	1	3
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32805
12	pbLineSpec	4	
16	cbLineSpec	2	
18	pbLineControl	4	
22	cbLineControl	2	
24	pbDialString	4	
28	cbDialString	2	
30	pLhRet	4	
34	sLhRet	2	2
36	pcbStringRet	4	
40	scbStringRet	2	2

TsDataRead (lh, cTimeout, pbData, cbDataMin, cbDataMax, pcbDataRet): ercType

Description

TsDataRead moves a block of received data from the modem to the area specified in the caller's memory. If the time interval expires before the memory buffer is full, the service returns with status code 11289 ("Data timeout").

If a parity error occurs, the last byte returned will contain the error.

Procedural Interface

TsDataRead (lh, cTimeout, pbData, cbDataMin, cbDataMax, pcbDataRet): ercType

where

lh

is the line handle (word) returned by a TsDataOpenLine call.

cTimeout

is the maximum amount of time (word) in units of 100ms that the caller wants to wait before returning the call with a timeout error.

pbData

describes the memory area to which the data is returned. The size of the memory area must be large enough for at least *cbDataMax* bytes.

cbDataMin

is a word specifying the minimum number (must be at least 1) of bytes that will be returned (unless an error occurs).

cbDataMax

is a word specifying the maximum number of bytes that can be returned into the memory area. *cbDataMax* must not be smaller than *cbDataMin*.

pcbDataRet

is the memory address of a word to which the actual count of bytes read will be returned.

Request Block

scbDataRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	8
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32806
12	lh	2	
14	cTimeout	2	
16	cbDataMin	2	
18	pbData	4	
22	cbDataMax	2	
24	pcbDataRet	4	
28	scbDataRet	2	2

TsDataRetrieveParams (lh, pStatusRet, sStatusRetMax): ercType

Description

TsDataRetrieveParams returns the parameters of the specified line to the caller.

Procedural Interface

TsDataRetrieveParams (lh, pStatusRet, sStatusRetMax): ercType

where

lh

is the line handle (word) returned by TsDataOpenLine.

pStatusRet
sStatusRetMax

describe the memory area where the status structure is to be returned. (For the format of this structure, see "Data Control Structure" in Chapter 4, "System Structures.")

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	8
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32811
12	lh	2	
14	pStatusRet	4	
18	sStatusRetMax	2	

TsDataUnacceptCall (pbLineSpec, cbLineSpec): ercType

Description

TsDataUnacceptCall retrieves 2 previously submitted TsDataOpenLine requests before the timeout expires.

This request is useful for a program that wants to change control parameters as the result of user input, or for a program that only wants to accept calls when some other condition, such as time of day, is satisfied.

The TsDataOpenLine request will be returned with status code 11283 ("TsDataUnacceptCall issued").

If no TsDataOpenLine request is waiting at the service, TsDataUnacceptCall will return status code 11284 ("No TsDataOpenLine request").

Procedural Interface

TsDataUnacceptCall (pbLineSpec, cbLineSpec): ercType

where

pbLineSpec
cbLineSpec

describe the line specification used in the TsDataOpenLine call. (For details, see the description of TsDataOpenLine.)

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	0
1	RtCode	1	1
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32812
12	pbLineSpec	4	
16	cbLineSpec	2	

TsDataWrite (lh, cTimeout, pbData, cbData, pcbDataRet): ercType

Description

TsDataWrite writes a data block from the caller's memory to the modem.

Procedural Interface

TsDataWrite (lh, cTimeout, pbData, cbData, pcbDataRet): ercType

where

lh

is the line handle (word) returned by an TsDataReadOpenLine call.

cTimeout

is the maximum amount of time (word) in units of 100 ms that the caller wants to wait to transfer the data block over the modem. If the Telephone Service cannot transfer the data in this time period, it returns a timeout error to the caller.

pbData
cbData

describe the caller's memory area that contains the data.

pcbDataRet

is the memory address of a word to which the actual count of bytes written will be returned.

Request Block

scbDataRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	8
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32807
12	lh	2	
14	cTimeout	2	
16	pbData	4	
20	cbData	2	
22	pcbDataRet	4	
26	scbDataRet	2	2

TsDeinstall(iVpModule, pPhRet, erc): ercType

Description

TsDeinstall is issued by a program to deinstall the Telephone Service. The Telephone Service will unserve all of its request codes, respond to any outstanding requests with the specified status code, clean up its internal state, place the Voice Processor module in power–off state, and then respond to the TsDeinstall request with its partition handle.

Procedural Interface

TsDeinstall(iVpModule, pPhRet, erc): ercType

where

iVpModule

is the Voice Processor module (word), 1 being the first Voice Processor module to the right of the CPU module.

pPhRet

is the memory address of a word to which the Telephone Service partition handle will be returned.

erc

is the status code (word) to be returned to any client programs of the Telephone Service.

Request Block

sPhRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32824
12	iVpModule	2	
14	erc	2	
16	pPhRet	4	
20	sPhRet	2	2

TsDial (iVpModule, line, pbDialString, cbDialString, cErrorTimeout, pcbStringRet): ercType

Description

TsDial causes the dual-tone multifrequency (DTMF) encoder to generate the specified characters. The DTMF generator is automatically connected to the specified line.

Procedural Interface

TsDial (iVpModule, line, pbDialString, cbDialString, cErrorTimeout, pcbStringRet): ercType

where

iVpModule

is the Voice Processor module (word), 1 being the first Voice Processor module to the right of the CPU module.

line

is the telephone line (word) to be connected, 0 for line 1 and 1 for line 2.

pbDialString
cbDialString

describe the number (character string) to be dialed. For a description of the characters that can be contained in this string, see the description of *pbDialString/cbDialString* in the TsDataOpenLine operation.

cErrorTimeout

is the maximum time (word) in units of 100 ms that is allowed before returning an error when waiting for a dial tone.

pcbStringRet

is the memory address of a return status structure that contains the number of characters processed.

Request Block

scbStringRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32815
12	iVpModule	2	
14	line	2	
16	cErrorTimeout	2	
18	pbString	4	
22	cbString	2	
24	pcbStringRet	4	
28	scbStringRet	2	2

TsDoFunction (iVpModule, function): ercType

Description

TsDoFunction causes the Telephone Service to perform functions similar to those available by pressing buttons on a typical two-line telephone unit.

Procedural Interface

TsDoFunction (iVpModule, function): ercType

where

iVpModule

is the Voice Processor module (word), 1 being the first Voice Processor module to the right of the CPU module.

function

is the function (word) to be done. Note that functions 0 through 19 are performed by the Voice Processor, while functions 20 through 32 are performed by the Audio Processor. The word values of *function* are described below:

Value	Description
0	Selects line 1 and connects to it, placing other line on hold if it was connected.
1	Selects line 2 and connects to it, placing other line on hold if it was connected.
2	Selects line 1 and connects to it, hanging up other line if it was connected.
3	Selects line 2 and connects to it, hanging up other line if it was connected.
4	Places selected line on hold.
5	Hangs up selected line.

Value	Description
6	Connects the telephone unit to the selected line.
7	Disconnects the telephone unit and hangs up the line(s) attached to it.
8	Connects lines 1 and 2 together.
9	Disconnects lines and places the unselected line on hold.
10	Switches the telephone unit into monitor mode.
11	Switches the telephone unit into normal mode.
12	Locks the CODEC for the exclusive use of caller.
13	Unlocks the CODEC.
14	Sets the switch so that the calling application is brought forward whenever the telephone unit goes offhook.
15	Resets the Context Manager switch.
16	Resets dialing.
17	Resets the detector.
18	Resets the modem.
19	Resets the Voice Processor module.
20	Starts Digital Signal Processor (DSP) code execution.
21	Stops DSP code execution by performing a Reset.
22	Reserved.
23	Reserved.

Value	Description
24	Allows input from the Audio Summing Line to the encoder/decoder.
25	Allows input from the microphone to the encoder/decoder.
26	Turns on the diagnostic loopback.
27	Turns off the diagnostic loopback.
28	Allows input from the Audio Summing Line to the motherboard speaker.
29	Disallows input from the Audio Summing Line to the motherboard speaker.
30	Allows output from the encoder/decoder to the motherboard speaker.
31	Disallows output from the encoder/decoder to the motherboard speaker.
32	Turns on the encoder/decoder.
33	Turns off the encoder/decoder.
34	Resets the table of loaded COFF sections.
35	Turns on the digital passthrough.
36	Turns off the digital passthrough.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32817
12	iVpModule	2	
14	function	2	

TsGetStatus (iVpModule, pStatusRet, sStatusRetMax, fNoWait): ercType

Description

TsGetStatus returns the state of the hardware and all users.

This operation is used to determine, for instance, the state of the telephone lines, the telephone hand set, and the cross point switch. The information is returned in the Telephone Status Structure. For a description of this structure, see "Telephone Status Structure" in Chapter 4, "System Structures."

Procedural Interface

TsGetStatus (iVpModule, pStatusRet, sStatusRetMax, fNoWait): ercType

where

iVpModule

is the Voice Processor module, 1 being the first Voice Processor module to the right of the CPU module.

pStatusRet
sStatusRetMax

describe the memory area where the Telephone Status Structure is to be returned.

fNoWait

if TRUE, the status will be returned immediately. If FALSE the status will be returned the next time something changes.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32813
12	iVpModule	2	
14	fNoWait	2	
16	pStatusRet	4	
20	sStatusRetMax	2	

TsHold (iVpmodule, line): ercType

Description

TsHold causes the specified telephone line to be disconnected from all devices and placed on hold.

Procedural Interface

TsHold (iVpModule, line): ercType

where

iVpModule

is the Voice Processor mdule, 1 being the first Voice Processor module to the right of the CPU module.

line

is either 0 or 1 to specify the telephone line 1 or 2.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32820
12	iVpModule	2	
14	line	2	

TsLoadCallProgressTones (iModule, pbFilename, cbFilename): ercType

Description

TsLoadCallProgressTones provides call progress tone definitions which are used by a subsequent TsDial operation to interpret call progress tones for the current telephone network.

A call progress tone is any tone that occurs during a telephone call. Examples include dial tones, busy signals, and ring tones.

The application provides a configuration file that contains the characteristics of each call progress tone. (For a description of this file, see "Voice/Data Services" the *CTOS Programming Guide*.)

Unless TsDial is aware of the traits for each call progress tone, it cannot successfully interpret call progress tones after it dials a number. Consequently, if the current telephone line does not use standard U.S. telephone signals, an application may need to call TsLoadCallProgressTones before it calls TsDial.

Status code 11215 ("File incomplete") is returned if the tone configuration file is damaged or incomplete. Status code 11214 ("File not found") is returned if the configuration file is not found. Status code 11203 ("Invalid request parameter") is returned if any parameters are invalid.

This operation is performed by the Telephone Service. (See "Voice/Data Services" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

TsLoadCallProgressTones (iModule, pbFilename, cbFilename): ercType where

iModule

is the Voice Processor module, 1 being the first Voice Processor module to the right of the CPU module.

pbFilename
cbFilename

describe a valid CTOS file specification. This file contains the configuration values to be loaded.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33201
12	iModule	2	
14	pbFilename	4	
18	cbFilename	2	

TsOffHook (iVpModule, pThRet, line): ercType

Description

TsOffHook makes the specified line go off hook. It does not change any connections of the cross point switch.

This operation returns a handle that can be used for adding devices to the line with the TsConnect operation.

Procedural Interface

TsOffHook (iVpModule, pThRet, line): ercType

where

iVpModule

is the Voice Processor module, 1 being the first Voice Processor module to the right of the CPU module.

pThRet

is the memory address of a word to which a telephone connection handle will be returned.

line

is either 0 or 1 to specify telephone line 1 or 2.

Request Block

sThRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32818
12	iVpModule	2	
14	line	2	
16	pThRet	4	
20	sThRet	2	2

TsOnHook (iVpModule, line): ercType

Description

TsOnHook causes the specified line to be disconnected from all devices and go onhook or "hungup".

Procedural Interface

TsOnHook (iVpModule, line): ercType

where

iVpModule

is the Voice Processor module, 1 being the first Voice Processor module to the right of the CPU module.

line

is either 0 or 1 to specify telephone line 1 or 2.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32819
12	iVpModule	2	
14	line	2	

TsQueryConfigParams (iVpModule, pConfigRet, sConfigRetMax, pbFileSpecRet, cbFileSpecRetMax, pcbFileSpecRet): ercType

Description

TsQueryConfigParams is used to get the Telephone Service configuration file name and the current configuration information.

Procedural Interface

TsQueryConfigParams (iVpModule, pConfigRet, sConfigRetMax, pbFileSpecRet, cbFileSpecRetMax, pcbFileSpecRet): ercType

where

iVpModule

is the Voice Processor module, 1 being the first Voice Processor module to the right of the CPU module.

pConfigRet
sConfigRetMax

describe the memory area where the Telephone Service Configuration Block is returned. (For the format of this structure, see "Telephone Service Configuration Block" in Chapter 4, "System Structures.")

pbFileSpecRet
cbFileSpecRetMax

describe the memory area where the file specification for the Telephone Service configuration file is returned.

pcbFileSpecRet

is the memory address of a word to which the actual size of the file specification is returned.

Request Block

scbFileSpecRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	3
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32821
12	iVpModule	2	
14	pConfigRet	4	
18	sConfigRetMax	2	
20	pbFileSpecRet	4	
24	cbFileSpecRetMax	2	
26	pcbFileSpecRet	4	
30	scbFileSpecRet	2	2

TsReadTouchTone (iVpModule, device, pbDigits, cbDigitsMax, pcbDigitsRet, bStopDigit, cTimeout): ercType

Description

TsReadTouchTone is used to read dual-tone multifrequency (DTMF) tones from a telephone line or the telephone unit.

Procedural Interface

TsReadTouchTone (iVpModule, device, pbDigits, cbDigitsMax, pcbDigitsRet, bStopDigit, cTimeout): ercType

where

iVpModule

is the Voice Processor module, 1 being the first Voice Processor module to the right of the CPU module.

device

specifies the telephone line or telephone unit. The values of *device* are

Value	**Description**
0	telephone line 1
1	telephone line 2
2	telephone unit

pbDigits
cbDigitsMax

describe the memory area where the digits are returned.

pcbDigitsRet

is the memory address of a word to which the actual number of digits read is returned.

bStopDigit

is a character that will terminate the input. The character may be 0 to 9, *, #, or A to D. Any other character is interpreted as meaning there is no stop digit.

cTimeout

is the time in units of 100ms after which the read will be terminated if there are no characters received.

Request Block

scbDigitsRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	8
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32816
12	iVpModule	2	
14	device	2	
16	bStopDigit	2	
18	cTimeout	2	
20	pbDigits	4	
24	cbDigitsMax	2	
26	pcbDigitsRet	4	
30	scbDigitsRet	2	2

This page intentionally left blank

TsRing (iVpModule, hz): ercType

Description

TsRing turns on (or off) the monitor's ring-like tone, allowing an application or end user to select the ring frequency interactively. If either telephone line rings or if the telephone unit goes offhook, monitor ringing returns to normal.

Procedural Interface

TsRing (iVpModule, hz): ercType

where

iVpModule

is the Voice Processor module on the X-Bus, 1 being the first Voice Processor module to the right of the CPU module.

hz

is the frequency (byte) to be used. A value of 0 means no ringing. The range of values that produces the ring-like tone is 1 to 255.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	3
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32823
12	iVpModule	2	
14	hz	1	

TsSetConfigParams (iVpModule, pConfig, sConfig): ercType

Description

TsSetConfigParams is used to set the Telephone Service configuration information.

Procedural Interface

TsSetConfigParams (iVpModule, pConfig, sConfig): ercType

where

iVpModule

is the Voice Processor module on the X-Bus, 1 being the first Voice Processor module to the right of the CPU module.

pConfig
sConfig

describe the Telephone Service Configuration Block. (For the format of this structure, see "Telephone Service Configuration Block" in Chapter 4, "System Structures.")

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32822
12	iVpModule	2	
14	pConfig	4	
18	sConfig	2	

TsVersion (iVpModule, pVerStruct, sVerStruct, pcbRet): ercType

Description

TsVersion returns a structure that contains the version of the installed Voice/Data Services.

Procedural Interface

TsVersion (iVpModule, pVerStruct, sVerStruct, pcbRet): ercType

where

iVpModule

is the Voice Processor module (word) on the X-Bus, 1 being the first Voice Processor module to the right of the CPU module.

pVerStruct
sVerStruct

describe a memory area to which the structure containing the version number is written. The format is as follows:

Offset	**Field**	**Size (Bytes)**
0	version	2

pcbRet

describes a word into which the length of the returned structure is placed.

Request Block

scbRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8216
12	iVpModule	2	
14	reserved	4	
18	pVerStruct	4	
22	sVerStruct	2	
24	pcbRet	4	
28	scbRet	2	2

TsVoiceConnect (iVpModule, fTelephoneUnit, fLine0, fLine1): ercType

Description

TsVoiceConnect specifies the connection to be used with a TsVoicePlaybackFromFile or TsVoiceRecordToFile operation in which the flag *fAutoStart* in the Voice Control Structure is FALSE. (See "Voice Control Structure" in Chapter 4, "System Structures.")

Before the call to TsVoiceConnect, use the TsDoFunction operation to lock the CODEC.

Procedural Interface

TsVoiceConnect (iVpModule, fTelephoneUnit, fLine0, fLine1): ercType

where

iVpModule

is the Voice Processor module on the X-Bus, 1 being the first Voice Processor module to the right of the CPU module.

fTelephoneUnit

if TRUE the CODEC will be connected to the telephone unit.

fLine0

if TRUE the CODEC will be connected to line 0.

fLine1

if TRUE the CODEC will be connected to line 1.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	5
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32803
12	iVpModule	2	
14	fTelephoneUnit	1	
15	fLine0	1	
16	fLine1	1	

TsVoicePlaybackFromFile (iVpModule, pWorkArea, sWorkArea, pVoiceControl, sVoiceControl, pLfaLast, pqSampleLast, pStatusRet): ercType

Description

TsVoicePlaybackFromFile plays back voice data from the specified file or from memory.

(For a description of how to use TsVoicePlaybackFromFile to play back from memory, see the *CTOS Programming Guide*.)

Procedural Interface

TsVoicePlaybackFromFile (iVpModule, pWorkArea, sWorkArea, pVoiceControl, sVoiceControl, pLfaLast, pqSampleLast, pStatusRet): ercType

where

iVpModule

is the Voice Processor module on the X-Bus, 1 being the first Voice Processor module to the right of the CPU module.

pWorkArea
sWorkArea

describe the memory area to be used by the Telephone Service for communicating with the Voice Processor module. *sWorkArea* must be at least 13K bytes. This value can be larger but must be expressed in 1K byte increments.

pVoiceControl
sVoiceControl

describe the Voice Control Structure. (For the format of this structure, see "Voice Control Structure" in Chapter 4, "System Structures.")

pLfaLast

is the memory address of the logical file address (lfa) returned when playback ended.

pqSampleLast

is the memory address of the sample number returned when playback ended.

pStatusRet

is the memory address of the return status of the TsVoicePlaybackFromFile operation. The status values are described below:

Value	**Description**
0	termination because of an error
1	termination because the TsVoiceStop command was issued
2	termination because the telephone line or telephone unit was placed onhook
3	termination because the maximum lfa (specified in the Voice Control Structure) was reached
4	termination because the maximum sample (specified in the Voice Control Structure) was reached

Request Block

sLfaLast and *sqSampleLast* are always 4. *sStatusRet* is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	3
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32801
12	iVpModule	2	
14	pWorkArea	4	
18	sWorkArea	2	
20	pVoiceControl	4	
24	sVoiceControl	2	
26	pLfaLast	4	
30	sLfaLast	2	4
32	pqSampleLast	4	
36	sqSampleLast	2	4
38	pStatusRet	4	
42	sStatusRet	2	2

This page intentionally left blank

TsVoiceRecordToFile (iVpModule, pWorkArea, sWorkArea, pVoiceControl, sVoiceControl, pLfaNext, pqSampleNext, pStatusRet): ercType

Description

TsVoiceRecordToFile records voice data from a telephone line or the telephone unit to the specified file.

Procedural Interface

TsVoiceRecordToFile (iVpModule, pWorkArea, sWorkArea, pVoiceControl, sVoiceControl, pLfaNext, pqSampleNext, pStatusRet): ercType

where

iVpModule

is the Voice Processor module on the X-Bus, 1 being the first Voice Processor module to the right of the CPU module.

pWorkArea
sWorkArea

describes a memory location to be used by the Telephone Service for communicating with the Voice Processor module. *sWorkArea* must be at least 13K bytes. This value can be larger but must be expressed in 1K byte increments.

pVoiceControl
sVoiceControl

describe the Voice Control Structure. (For the format of this structure, see "Voice Control Structure" in Chapter 4, "System Structures.")

pLfaNext

is the memory address of the returned logical file address (lfa) of the next sector in the recording file at termination.

pqSampleNext

is the memory address of the returned sample number, which would have been given the next sample in the recording at termination.

pStatusRet

is the memory address of the return status of the TsVoiceRecordToFile operation. The status values are described below:

Value	**Description**
0	termination because of an error
1	termination because the TsVoiceStop command was issued
2	termination because the telephone line or the telephone unit was placed onhook
3	termination because the maximum lfa (specified in the Voice Control Structure described in Chapter 4, "System Structures") was reached
4	termination because the maximum sample (specified in the Voice Control Structure) was reached
5	termination because the maximum pause (specified in the Voice Control Structure) was detected

Request Block

sLfaNext and *sqSampleNext* are always 4. *sStatusRet* is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	3
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32802
12	iVpModule	2	
14	pWorkArea	4	
18	sWorkArea	2	
20	pVoiceControl	4	
24	sVoiceControl	2	
26	pLfaNext	4	
30	sLfaNext	2	4
32	pqSampleNext	4	
36	sqSampleNext	2	4
38	pStatusRet	4	
42	sStatusRet	2	2

This page intentionally left blank

TsVoiceStop (iVpModule): ercType

Description

TsVoiceStop terminates voice recording or playback.

Procedural Interface

TsVoiceStop (iVpModule): ercType

where

iVpModule

is the Voice Processor module on the X-Bus, 1 being the first Voice Processor module to the right of the CPU module.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32804
12	iVpModule	2	

This page intentionally left blank

TurnOffMailNotification: ercType

Description

TurnOffMailNotification resets the environment so that subsequent calls to QueryMail will not notify the user of incoming mail. Signon.run calls this operation when invoked with the command Logout. This operation is the complement of the SetUpMailNotification.

Procedural Interface

TurnOffMailNotification: ercType

Request Block

TurnOffMailNotification is an object module procedure.

This page intentionally left blank

ULCmpB (prgbString1, prgbString2, cbString): WORD

Description

ULCmpB is a string comparison operation that is identical in format to NlsULCmpB except that NlsULCmpB also accepts the memory address of nationalized character sets.

Programs should use NlsULCmpB instead of ULCmpB for ease in nationalization. (For details, see "Native Language Support" in the *CTOS Operating System Concepts Manual.*)

ULCmpB returns 0FFFFh if two strings are the same. Otherwise, it returns the index of the first byte in the strings that is different. Uppercase and lowercase are ignored. For example, if the "Exec.Run" and "exec.run" strings are compared, the operation returns 0FFFFh. If the "Exec.run" and "Editor.run" strings are compared, the operation returns 1.

Procedural Interface

ULCmpB (prgbString1, prgbString2, cbString): WORD

where

prgbString1
prgbString2

are the memory addresses of two equal length strings.

cbString

is the length of the two strings.

Request Block

ULCmpB is a system–common procedure.

UndeleteFile (fhbNum, pbFileSpec, cbFileSpec, pbPassword, cbPassword): ercType

Caution: *This operation is for* **restricted use only.**

Description

UndeleteFile recovers a file that has already been deleted from a volume. The DeleteFile operation only deletes the memory addresses of the file data so that it cannot be accessed. The actual data still remains on the disk. UndeleteFile recreates all the memory addresses on the volume control structures so that they point to the file data, thereby recovering the deleted file.

Procedural Interface

UndeleteFile (fhbNum, pbFileSpec, cbFileSpec, pbPassword, cbPassword): ercType

where

fhbNum

is the file header page number (word) in the File Header Block for the file.

pbFileSpec
cbFileSpec

describe a character string specifying the name of the file to be undeleted.

pbPassword
cbPassword

describe a character string specifying a password that authorizes the requested file access.

Request Block

UndeleteFile is an object module procedure.

UnlockInContext (userNum): ercType

Description

UnlockInContext removes the inability to be swapped to disk (locked–in condition) previously assigned to the specified user number (partition) with the LockInContext operation. Specifying a user number of 0 causes the caller's partition to be unlocked. When all lock–ins previously asserted using LockInContext have been removed using UnlockInContext, the user number again becomes swappable.

Procedural Interface

UnlockInContext (userNum): ercType

where

userNum

specifies the user number (word) to be unlocked–in.

Request Block

UnlockInContext is a system-common procedure.

This page intentionally left blank

UnlockVideo

Caution: *For compatibility on all workstations, it is recommended that you use the Video Access Method (VAM) rather than using UnlockVideo.*

Description

UnlockVideo is used after calling LockVideo to remove a lock on the video structures used by the operating system.

If a program calls UnlockVideo without first calling LockVideo, it will be terminated with status code 508 ("Too many unlocks").

Procedural Interface

UnlockVideo

Request Block

UnlockVideo is a system-common procedure.

This page intentionally left blank

UnlockVideoForModify

Caution: *For compatibility on all workstations, it is recommended that you use the Video Access Method (VAM) rather than using UnlockVideoForModify.*

Description

UnlockVideoForModify is used after calling LockVideoForModify to remove a lock on the video structures used by the operating system.

If a program calls UnlockVideoForModify without first calling LockVideoForModify, a system crash will occur with status code 508 ("Too many unlocks").

This operation should not be used by programs that run under a partition-managing program, such as the Context Manager.

Procedural Interface

UnlockVideoForModify

Request Block

UnlockVideoForModify is a system-common procedure.

This page intentionally left blank

UnlockXbis: ercType

Caution: *This operation works on workstation hardware only. Do not use it on shared resource processor hardware.*

Description

UnlockXbis frees the XBIS structure for use by other programs.

Procedural Interface

UnlockXbis: ercType

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	0
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32798*

*On BTOS II, 3.2 workstation operating systems, the request code 8203 is also supported.

This page intentionally left blank

UnmapBusAddress (busAddress): ercType

NOTE: This operation is for use only by programs executing in protected mode.

Description

UnmapBusAddress releases any system hardware resources that were committed by an earlier call to the MapBusAddress or ReserveBusAddress operations.

On some systems it is not necessary for mapping hardware to be programmed to decode bus addresses. In such cases, UnmapBusAddress does nothing and returns a status code of 0.

On systems that use bus address mapping hardware (for example, the shared resource processor GP board), each call to MapBusAddress or ReserveBusAddress must be matched by a complementary call to UnmapBusAddress before a program exits or a system service deinstalls. This is necessary because bus addresses are a finite hardware resources that could run out, causing the system to become inoperable until rebooted.

Procedural Interface

UnmapBusAddress (busAddress): ercType

where

busAddress

is the a bus address (double word) previously obtained from the MapBusAddress or ReserveBusAddress operations.

Request Block

UnmapBusAddress is a system-common procedure.

UnmapPhysicalAddress (qAddr, qSizeOfRange, fSelector): ercType

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

UnmapPhysicalAddress deallocates the selector and/or linear address that was previously allocated by the MapPhysicalAddress operation.

Status code 473 ("Unknown linear address") is returned if the specified linear address is not mapped to a physical address.

Procedural Interface

UnmapPhysicalAddress (qAddr, qSizeOfRange, fSelector): ercType

where

qAddr

is either the selector to be deallocated (in the hight-order word), or the 32-bit linear address to be deallocated.

qSizeOfRange

is the number of bytes in the range of addresses specified in a previous call to MapPhysicalAddress.

fSelector

is a flag that is TRUE if *qAddr* contains a selector. This flag is FALSE if *qAddr* is a linear address.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	10
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	468
12	fSelector	2	
14	qAddr	4	
18	qSizeOfRange	4	

UnmarkQueueEntry (pbQueueName, cbQueueName, qeh): ercType

Description

UnmarkQueueEntry is used by the queue server to reset the specified queue entry as being unmarked (not in use). If the queue entry specifies a repeating time interval, the entry also is rescheduled. The queue entry to be unmarked is identified by a 32-bit queue entry handle that was previously placed in the Queue Status Block (*qehRet* field) by the MarkKeyedQueueEntry or the MarkNextQueueEntry operation. (For the format of the Queue Status Block, see Chapter 4, "System Structures.")

Procedural Interface

UnmarkQueueEntry (pbQueueName, cbQueueName, qeh): ercType

where

pbQueueName
cbQueueName

describe a queue name corresponding to a queue name specified when the queue was installed.

qeh

is the 32-bit queue entry handle.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	144
12	qeh	4	
16	pbQueueName	4	
20	cbQueueName	2	

UnsuspendProcess (pid): ercType

Description

UnsuspendProcess releases the process identified by the specified pid from the suspended state previously asserted with the SuspendProcess operation. The pid must be a valid process ID such as that returned by QueryProcessNumber. When all suspensions previously asserted using SuspendProcess have been deasserted by calls to UnsuspendProcess, the process can again be run.

Procedural Interface

UnsuspendProcess (pid): ercType

where

pid

specifies the process ID of the process to be released from suspension.

Request Block

UnsuspendProcess is a Kernel primitive.

This page intentionally left blank

UnsuspendUser (userNum): ercType

Description

UnsuspendUser releases all processes of the specified user number (partition) from the suspended state previously asserted with the SuspendUser operation. Specifying a value of 0 for *userNum* causes all processes in the caller's partition to be released. When all suspensions previously asserted using SuspendUser have been deasserted by calls to UnsuspendUser, the user number's processes can again be run.

Procedural Interface

UnsuspendUser (userNum): ercType

where

userNum

specifies the user number (word) whose processes are to be released from suspension.

Request Block

UnsuspendUser is a Kernel primitive.

This page intentionally left blank

UnzoomBox (pBoxDesc): ercType

Description

UnzoomBox collapses a box drawn with the ZoomBox operation. If ZoomBox saved the original character map in the box descriptor, UnzoomBox restores the display. (For details on the box descriptor, see the description of the ZoomBox operation.)

Procedural Interface

UnzoomBox (pBoxDesc): ercType

where

pBoxDesc

is the memory address of the box descriptor.

Request Block

UnzoomBox is an object module procedure.

This page intentionally left blank

UpdateOverlayLru (pProcInfo)

NOTE: This operation does not return a status code.

Description

UpdateOverlayLru is called from within one overlay to prevent any other overlay from being swapped out of memory in a program using the Virtual Code facility.

UpdateOverlayLru adjusts the apparent age of the overlay that it wants to retain in memory by updating the time of its most recent use so that it appears to have zero age.

UpdateOverlayLru and MakeRecentlyUsed are similar in that they override the Virtual Code facility's default replacement algorithm, which swaps out overlays based on age in memory. Neither operation, however, guarantees that the overlay will be permanently resident in memory. Use MakePermanent or MakePermanentP if your intent is to keep an overlay permanently resident in memory.

MakeRecentlyUsed differs from UpdateOverlayLru in preventing the calling overlay from being swapped out. (See MakeRecentlyUsed.)

Procedural Interface

UpdateOverlayLru (pProcInfo)

where

pProcInfo

is the memory address of the overlay whose age is to be updated.

Request Block

UpdateOverlayLru is an object module procedure.

VacatePartition (userNumPartition): ercType

Description

VacatePartition terminates the program in the application partition specified by the user number but does not load and activate the exit run file. VacatePartition leaves the partition vacant.

If the partition is locked, a status code is returned and a flag is set in the Application System Control Block to notify the program in the partition of the attempted VacatePartition operation.

Procedural Interface

VacatePartition (userNumPartition): ercType

where

userNumPartition

is the user number (word) returned from a CreatePartition or GetPartitionHandle operation.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	180
12	userNumPartition	2	

Wait (exchange, ppMsgRet): ercType

Description

The Wait primitive checks whether messages are queued at the specified exchange. If messages are queued, then the message that was queued first is removed from the queue and its memory address is returned to the calling process; the calling process then continues execution.

If no messages are queued, then the Process Control Block of the calling process is queued at the exchange and the process is placed into the waiting state. In the waiting state, the process stops executing and relinquishes control of the processor. The calling process remains in the waiting state until another process queues a message at the specified exchange using the Send, PSend, Request, or Respond primitives. The calling process then leaves the waiting state and is placed into the ready state. The memory address of the message queued at the exchange by the other process is returned to the calling process, which resumes execution when it becomes the highest priority ready process.

Procedural Interface

Wait (exchange, ppMsgRet): ercType

where

exchange

is the identification of the exchange at which to wait.

ppMsgRet

is the memory address of a 4-byte field where the memory address of the message, if any, that was queued first at the exchange is returned.

Request Block

Wait is a Kernel primitive.

WaitLong (exchange, ppMsgRet): ercType

Description

The WaitLong primitive is similar to Wait but should be used in lieu of Wait if it is anticipated that the calling process will be waiting for a long time, that is, for more than 30 seconds.

WaitLong differs from Wait in that the operating system memory manager is notified that the calling process is a good candidate for being swapped out if the memory that it is occupying is required for some other reason.

WaitLong checks whether messages are queued at the specified exchange. If messages are queued, then the message that was queued first is removed from the queue and its memory address is returned to the calling process; the calling process then continues execution.

If no messages are queued, then the Process Control Block of the calling process is queued at the exchange and the process is placed into the waiting state. In the waiting state, the process stops executing and relinquishes control of the processor. The calling process remains in the waiting state until another process queues a message at the specified exchange using the Send, PSend, Request, or Respond primitives. The calling process then leaves the waiting state and is placed into the ready state. The memory address of the message queued at the exchange by the other process is returned to the calling process, which resumes execution when it becomes the highest priority ready process.

Procedural Interface

Wait (exchange, ppMsgRet): ercType

where

exchange

is the identification of the exchange at which to wait.

ppMsgRet

is the memory address of a 4-byte field where the memory address of the message, if any, that was queued first at the exchange is returned.

Request Block

WaitLong is a Kernel primitive.

WhereTerminalBuffer has no procedural interface. You must make a request block and issue the request using Request, RequestDirect, or RequestRemote.

Caution: *This operation works on the shared resource processor hardware only. Do not use it on workstation hardware.*

Description

This request is used by the client to locate the terminal output buffer. (See Chapter 4, "System Structures," for the format of the terminal output buffer.) The returned address of the output buffer is in linear format. In linear format, word ordering and byte ordering within each word are exactly the opposite to the Intel 80x86 processor convention, which places the most significant byte at the highest address.

Procedural Interface

WhereTerminalBuffer has no procedural interface. You must make a request block and issue the request using one of the request operations listed above

where

bDestCPU

is the slot number (byte) of the Cluster or Terminal Processor to which the asychronous terminal is connected.

bSourceCpu

is the slot number (byte) of the client.

bPort

is the port number (byte) of a terminal attached to an RS-232-C port.

bWindow

is a byte value that must be 0.

ppOutputBufferRet
spOutputBufferRet

refer to the output buffer for a terminal that lives in the address space of *bDestCPU*. The returned memory address is in linear format. This linear pointer addresses the terminal output buffer. (See "Terminal Output Buffer" in Chapter 4, "System Structures.")

Request Block

spOutputBufferRet is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	ExchResp	2	
8	ercRet	2	
10	rqCode	2	12308
12	bDestCPU	1	
13	bSourceCPU	1	
14	bPort	1	
15	bWindow	1	
16	ppOutputBufferRet	4	
20	spOutputBufferRet	2	4

This page intentionally left blank

WildCardClose (pBuf): ercType

NOTE: An application should call WildCardClose after it has finished using the WildCardInit and WildCardNext operations.

Description

WildCardClose frees internal resources used by the WildCardInit and WildCardNext operations to match wild-card specifications with file names.

Procedural Interface

WildCardClose (pBuf): ercType

where

pBuf

  is the memory address of the buffer previously used by the WildCardInit and WildCardNext operations.

Request Block

WildCardClose is an object module procedure.

This page intentionally left blank

WildCardInit (pb, cb, pBuf, sBuf): ercType

Description

WildCardInit establishes a wild carded file specification to be used in subsequent calls to the WildCardNext operation. WildCardInit and WildCardNext still work correctly when given a file specification without wild card characters.

Procedural Interface

WildCardInit (pb, cb, pBuf, sBuf): ercType

where

pb
cb

describe the wild carded file specification.

pBuf
sBuf

describe a buffer of 800 bytes. This buffer is used in successive calls to the related operation WildCardNext.

The following is a description of the buffer pointed to by pBuf:

Offset	**Name**	**Length**
0	fWildcard	1
1	oFileHeaders	2
	. . .	

The returned value of *fWildcard* is either TRUE to signify that the string is a wild carded string, or FALSE to signify that the string contains no wild card characters. The value 65535 is returned at *oFileHeaders* if the file specification includes the name of a CD-ROM device.

Request Block

WildCardInit is an object module procedure.

WildCardMatch (pbWildStr, cbWildStr, pbStr, cbStr): Flagtype

Description

WildCardMatch checks a string against a wildcard specification. The operation returns TRUE if the string matches, FALSE if not.

Procedural Interface

WildCardMatch (pbWildStr, cbWildStr, pbStr, cbStr): Flagtype

where

pbWildStr
cbWildStr

describe a wildcard string.

pbStr
cbStr

describe a string to be checked.

Request Block

WildCardMatch is an object module procedure.

This page intentionally left blank

WildCardNext (pBuf, psdRet): ercType

Description

WildCardNext returns the next file name that matches a wild carded file specification supplied previously by a call to WildCardInit.

WildCardNext is called repetitively to process the next consecutive file name that matches the wild carded file specification until there are no more matching file names. Then it returns status code 1 ("End of file").

Procedural Interface

WildCardNext (pBuf, psdRet): ercType

where

pBuf

is the memory address of a buffer. This must be the same buffer used when calling WildCardInit.

The following is a description of the buffer pointed to by *pBuf*:

Offset	Name	Length
0	fWildcard	1
1	oFileHeaders	2
	. . .	

When WildCardNext returns, *oFileHeaders* is set to the sector number in FileHeaders.sys for the returned file specification. To create an lfa for the file header information for this file, multiply this value by 512 (the size of one sector). The value 65535 is returned at *oFileHeaders* if the file specification includes the name of a CD-ROM device.

psdRet

is the address of a 6 byte block of memory. The memory address of the next matching file specification is returned in the first 4 bytes, and its size is stored in the last 2 bytes.

Request Block

WildCardNext is an object module procedure.

Write (fh, pBuffer, sBuffer, lfa, psDataRet): ercType

Description

Write transfers an integral number of 512-byte sectors from memory to disk. Write returns only when the requested transfer is complete. WriteAsync and CheckWriteAsync are used to overlap computation and input/output transfer. Write can also be accessed as the WriteFile operation.

Write does not set the end-of-file pointer. (See SetFileStatus.) Attempting to write beyond the end of a file results in the return of status code 2 ("End of medium").

To accommodate programming languages in which Write is a reserved word, WriteFile is permitted as a synonym for the Write operation.

Procedural Interface

Write (fh, pBuffer, sBuffer, lfa, psDataRet): ercType

where

fh

is a file handle (word) returned from an OpenFile operation. The file must be open in modify mode.

pBuffer

is the memory address of the first byte of the buffer from which the data is to be written. The buffer must be word aligned.

sBuffer

is the count (word) of bytes to be written from memory. It must be a multiple of 512.

lfa

is the byte offset (double word), from the beginning of the file, of the first byte to be written. It must be a multiple of 512.

psDataRet

is the memory address of the word where the count of bytes successfully written is to be returned.

Request Block

DataRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	36
12	fh	2	
14	lfa	4	
18	pBuffer	4	
22	sBuffer	2	
24	psDataRet	4	
28	ssDataRet	2	2

WriteAsync (fh, pBuffer, sBuffer, lfa, pRq, exchangeReply): ercType

Description

WriteAsync initiates the transfer of an integral number of 512-byte sectors from memory to disk. CheckWriteAsync must be called to check the completion status of the transfer.

The information returned by Write with its *psDataRet* argument and ercType status is obtained by CheckWriteAsync. WriteAsync does not set the end-of-file pointer. (See SetFileStatus.)

Procedural Interface

WriteAsync (fh, pBuffer, sBuffer, lfa, pRq, exchangeReply): ercType

where

fh

is a file handle (word) returned from an OpenFile operation. The file must be open in modify mode.

pBuffer

is the memory address of the first byte of the buffer from which the data is to be written. The buffer must be word aligned.

sBuffer

is the count (word) of bytes to be written from memory. It must be a multiple of 512.

lfa

is the byte offset (double word), from the beginning of the file, of the first byte to be written. It must be a multiple of 512.

pRq

is the memory address of a 64-byte area to be used as workspace by WriteAsync.

exchangeReply

is an exchange provided by the client for the exclusive use of WriteAsync and CheckWriteAsync.

Request Block

WriteAsync and CheckWriteAsync are procedural interfaces to the Write operation. (See Write.)

WriteBsRecord (pBswa, pb, cb, pcbRet): ercType

Description

WriteBsRecord writes the specified count of bytes to the open output byte stream identified by the memory address of the Byte Stream Work Area from the specified memory area. Because output is buffered, there is no guarantee of the time at which output is actually written. Only the CheckpointBs and CloseByteStream operations ensure that data was actually written.

Procedural Interface

WriteBsRecord (pBswa, pb, cb, pcbRet): ercType

where

pBswa

is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

pb

is the memory address of the data to be written.

cb

is the count (word) of bytes to write.

pcbRet

is the memory address of the word where the count of data bytes successfully written is returned.

Request Block

WriteBsRecord is an object module procedure.

WriteByte (pBswa, b): ercType

Description

WriteByte writes one byte to the open output byte stream identified by the memory address of the Byte Stream Work Area. Because output is buffered, there is no guarantee of the time at which output is actually written. Only the CheckpointBs and CloseByteStream operations ensure that data was actually written.

Procedural Interface

WriteByte (pBswa, b): ercType

where

pBswa

  is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

b

  is the eight-bit byte to write.

Request Block

WriteByte is an object module procedure.

This page intentionally left blank

WriteByteStreamParameterC (pBs, wParamNum, wParam): ercType

Description

WriteByteStreamParameterC modifies the value of the specified communications parameter. (See "Communications Programming" in the *CTOS Operating System Concepts Manual.*)

This operation blocks (waits) while there are characters still in the transmit buffer. There is no asynchronous form of this operation, but to avoid actually having the calling process wait, CheckPointBsAsyncC can be called prior to WriteByteStreamParameterC.

The value of *wParamNum* is as follows:

Code	**Description**
0	Number of data bits (5-8)
1	Parity (0=none, 1=even, 2=odd, 3=one, 4=zero)
2	Baud Rate (word) (truncated to nearest legal value)
3	Stop Bits (1=1, 2=2)
4	Transmit time out (in seconds) (0000h=immediate) (FFFFh=infinite)
5	Receive time out (in seconds) (0000h=immediate) (FFFFh=infinite)
6	Carriage return/line feed mapping mode (0=binary, 1=new line)
7	New line mapping mode (0=binary, 1=carriage return, 2=carriage return/line feed)
8	Line Control (0=none, 1=XON/XOFF, 2=CTS, 3=both)
9	End of file (EOF) byte (low byte is EOF byte, high byte is TRUE when EOF checking is enabled, FALSE otherwise)
10	Receive baud rate (word)

Code	Description
11	Transmit baud rate (word)
12	Tab expansion size (byte; 0=treat tabs literally)
13	Maximum line length (byte; for auto-line-wrap; 0=disabled)

ParamNum 2 sets both receive and transmit baud rates.

ParamNum 10 or 11 can be used to set one or the other. (This is only supported on certain hardware; an error is returned otherwise.)

Procedural Interface

WriteByteStreamParameterC (pBs, wParamNum, wParam): ercType

where

pBs

is the memory address of the Byte Stream Work Area (BSWA).

wParamNum

is the parameter number to write (word).

wParam

is the value to be written (word).

Request Block

WriteByteStreamParameterC is an object module procedure.

WriteCommLineStatus (commLineHandle, wStatusMask, wStatus): ercType

Description

WriteCommLineStatus allows certain RS-232 signals, whose function is not defined by the serial communications controller (such as the 8274 controller), to be raised or lowered by the application program in a machine-independent fashion.

WriteCommLineStatus writes to the specified status bits, changing the condition of the corresponding status lines.

wStatusMask is a word with one of the following bit masks:

Mask	**Description**
0100h	SRTS (Secondary Request-to-Send)
0200h	RS (Rate Select)
0400h	STX (Secondary Transmit)
0800h	SS (Select Standby)
1000h	DTR (8530 DMA mode only)

Masks may be ORed together to reference more than one status bit at a time. Only the status bits selected by *wStatusMask* are accessed. The others are not written.

Procedural Interface

WriteCommLineStatus (commLineHandle, wStatusMask, wStatus): ercType

where

commLineHandle

is a handle (word) returned by InitCommLine.

wStatusMask

is a bit mask (word) that selects status lines affected.

wStatus

is a word containing the data values of the selected status bits, in the corresponding bit positions.

Request Block

WriteCommLineStatus is a system-common procedure.

WriteDaFragment (pDawa, qiRecord, pFragment, rbFragment, cbFragment): ercType

Description

WriteDaFragment writes a record fragment to the open DAM file identified by the memory address of the Direct Access Work Area. The written record fragment is specified by the record number, relative offset, and the byte count. The DAM file is automatically extended to accommodate new records.

Procedural Interface

WriteDaFragment (pDawa, qiRecord, pFragment, rbFragment, cbFragment): ercType

where

pDawa

is the memory address of the same Direct Access Work Area that was supplied to OpenDaFile.

qiRecord

is a 32-bit unsigned integer specifying the number of the record containing the record fragment to be written.

pFragment

is the memory address of the memory area from which the record fragment is written.

rbFragment

is the offset from the beginning of the record to the first byte of the record fragment.

cbFragment

is the size of the record fragment.

Request Block

WriteDaFragment is an object module procedure.

WriteDaRecord (pDawa, qiRecord, pRecord): ercType

Description

WriteDaRecord writes a record to the open DAM file identified by the memory address of the Direct Access Work Area. The written record is specified by the record number. The DAM file is automatically extended to accommodate new records.

WriteDaRecord can write a record with a record number larger than any existing record number. If this is done, the file is extended and standard record header and trailer formats are written automatically to all added sectors. The time required for the WriteDaRecord operation is proportional to the amount by which the file is extended.

Procedural Interface

WriteDaRecord (pDawa, qiRecord, pRecord): ercType

where

pDawa

is the memory address of the same Direct Access Work Area that was supplied to OpenDaFile.

qiRecord

is a 32-bit unsigned integer specifying the number of the record to be written.

pRecord

is the memory address of the memory area from which the record is written.

Request Block

WriteDaRecord is an object module procedure.

WriteHardId (wID): ercType

NOTE: Applications that need to run on older BTOS operating systems should use the OldWriteHardId operation.

Description

WriteHardId stores a value (1 to 126) for either an add-on hardware ID device on an X-bus workstation or the non-volatile RAM on a SuperGen or B39 workstation. The value of the hardware ID is saved even if the workstation is powered off.

On X-bus workstations, if there is no physical hardware ID device, status code 693 ("No device present on the I-bus") is returned.

Procedural Interface

WriteHardId (wID): ercType

where

wId

is a word value specifying the new hardware ID for the workstation.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCoce	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8193
12	wId	2	

WriteIBusDevice (wId, pbByteString, cbByteString): ercType

Description

WriteIBusDevice writes a byte string to a specific I–Bus device. The operating system surrounds the string with the appropriate I–Bus protocol bytes and sends the string to the device.

Procedural Interface

WriteIBusDevice (wId, pbByteString, cbByteString): ercType

where

wId

is the 2 byte I–Bus device identifier.

pbByteString
cbByteString

describe the byte string to be written to the I–Bus device.

Request Block

WriteIBusDevice is a system–common procedure.

This page intentionally left blank

WriteIBusEvent (prgIBusEvents, cbIBusEvents):ercType

Description

WriteIBusEvent places I-Bus events (such as keystrokes and mouse movement) in the queue for the current type-ahead buffer as set by the AssignKbd operation.

I-Bus events are 2-byte pairs (*bEventType* and *bData*) of the following three types:

1. An unencoded keystroke event requires the following 2-byte pair of data:

bEventType	**bData**
20h	indicates the keyboard code generated by an unencoded keyboard (for example, 68h is the value for the h key on CTOS keyboards)

2. A mouse X and Y position event occurs only with the combination of motion rectangle and mouse button events. A motion rectangle event requires the following five pairs of data:

bEventType	**bData**
60h	Mouse button code
81h	cursor X, low byte
81h	cursor X, high byte
82h	cursor Y, low byte
82h	cursor Y, high byte

3. A mouse button event also requires five pairs of data, as follows:

bEventType	bData
80h	Mouse button code
81h	cursor X, low byte
81h	cursor X, high byte
82h	cursor Y, low byte
82h	cursor Y, high byte

Bits 0 through 2 of the Mouse button code are interpreted as follows:

Bit	Value	Attribute
0	1	Right
1	2	Middle
2	4	Left

Cursor X and Y are normalized screen coordinates. As an example, for one workstation type the X and Y coordinates are (0, 0) for the top left corner of the screen and (32767, 22435) for the bottom right corner. (For details on screen coordinates, see the Mouse Services documentation in the *CTOS Programming Guide*.)

Procedural Interface

WriteIBusEvent (prgIBusEvents, cbIBusEvents):ercType

where

prgIBusEvents

is the address of an array of I-Bus events.

cbIBusEvents

is the size (word) in bytes of the I-Bus event array. Size is always is a multiple of 2.

Request Block

WriteIBusEvent is a system-common procedure.

This page intentionally left blank.

WriteKbdBuffer (eventType, userNum, mode, pBytesToWrite, sBytesToWrite, pcbRet): ercType

Caution: *This operation returns emulated key values. If an application is reading the keyboard in raw unencoded mode, it returns error code 604 ("Invalid argument to a keyboard operation").*

Description

WriteKbdBuffer writes unencoded keystrokes directly into the type-ahead buffer of an application. A special mode allows keyboard and mouse input to be disabled. This feature prevents keyboard or mouse-derived data from appearing in the type-ahead buffer during calls to WriteKbdBuffer.

WriteKbdBuffer allows a context managing program to take characters from one context, untranslate them into their unencoded values, and place these unencoded values into the type-ahead buffer of a different context.

To write mouse events, only the *bCharRet*, *wX*, and *wY* values of the ReadInputEvent operation are passed. The mouse server prepends the event type when its ProcessInputEvent request gets the other data out of the type-ahead buffer.

WriteKbdBuffer can also be used for application demonstrations. As an example, a word processor might show the steps required to print a file. During the demonstration, the end user would watch as the word processor writes words into a print command field.

In this example, the program would use WriteKbdBuffer to fill its own type-ahead buffer with the sample words. Subsequently, the program would write the contents of its type-ahead buffer to the video display.

Status code 619 ("No buffer space") is returned if all bytes do not fit in the type-ahead buffer of the application.

Procedural Interface

WriteKbdBuffer (eventType, userNum, mode, pBytesToWrite, sBytesToWrite, pcbRet): ercType

where

eventType

is the type of event associated with the bytes to be written to the type-ahead buffer. (See WriteIBusEvent for a description of event values.)

userNum

is the user number of the application whose type-ahead buffer is to receive the bytes.

mode

is the mode of the write operation. Each value indicates the following:

Value	Description
0	Write bytes into the type-ahead buffer until it is full. At *pcbRet*, WriteKbdBuffer returns the count of bytes successfully written.
1	If the type-ahead buffer is not large enough to hold all of the bytes, do not write any bytes. Instead, return status code 619 ("No buffer space"). At *pcbRet*, WriteKbdBuffer returns the amount of available space in the type-ahead buffer.
2	Disables input from the keyboard and mouse.
3	Enables input from the keyboard and mouse.

Value	Description
80h	For internal use only.
81h	For internal use only.

pBytesToWrite

is the memory address of the values that will be written to the type-ahead buffer.

sBytesToWrite

is the number of bytes to be written to the type-ahead buffer.

pcbRet

is the memory address of a word where WriteKbdBuffer returns either the available space (in bytes) in the type-ahead buffer or the count of bytes successfully written. The value returned depends upon the value of *mode*.

Request Block

scbRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	402
12	eventType	2	
14	userNum	2	
16	mode	2	
18	pBytesToWrite	4	
22	sBytesToWrite	2	
24	pcbRet	4	
28	scbRet	2	2

WriteLog (pbRecord, cbRecord): ercType

Description

WriteLog is used by an application program to write a variable-length record to the system Log File ([Sys]<Sys>Log.Sys). In the Executive, the **PLog** command is used to print the Log File. (See the *CTOS Executive Reference Manual* for details.)

You are allowed to use record type 0FFF7h only. The first two words of the record are interpreted by the **PLog** command as the record header: the first word contains the record type and the second word contains the length of the ASCII text (message) of the record. For example,

Description	**Size**	**Contents**
Record Type	word	0FFF7h
Record Length	word	18
ASCII Text	byte	This is a message.

In the example, *cbRecord* is 22.

When the record is written to the Log File, the operating system inserts additional information such as the date, hardware type, memory size, and the SignOn user name. (See the descriptions of the fixed offsets 0 through 19 in the Log file record format in Chapter 4, "System Structures.") The size of the Log file entry is the sum of the size of the record and the size of the additional information.

Log records are accumulated in a buffer within the operating system and are flushed to the log file when the buffer is full. Because this buffer flushing is an asynchronous operation, a WriteLog attempt may return with status code 290 ("Log buffer overflow"). In such a case, the process should be suspended temporarily and then retried. The Delay operation, for example, can be used to suspend execution. (For details, see the Delay operation.)

Procedural Interface

WriteLog (pbRecord, cbRecord): ercType

where

pbRecord
cbRecord

describe the record to be logged. Its maximum size is 128 bytes. cbRecord is the length of the record (that is, the count of bytes of the ASCII message plus 4).

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	125
12	reserved		6
18	pbRecord	4	
22	cbRecord	2	

WriteRemote has no procedural interface. You must make a request block and issue the request using Request, RequestDirect, or RequestRemote.

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases. It works only on shared resource processor hardware.*

Description

WriteRemote allows the caller to write to disk from the memory of any processor board on a shared resource processor.

Procedural Interface

WriteRemote has no procedural interface. You must make a request block and issue the request using one of the request operations listed above

where

fh

is a file handle (word) returned from an OpenFile operation. The file must be open in modify mode.

baBuffer

is the bus address of the first byte of the buffer from which the data is to be written. The buffer must be word aligned.

sBuffer

is the count (word) of bytes to be written from memory. It must be a multiple of the sector size (128, 256, 512, or 1024).

lfa

is the byte offset (double word), from the beginning of the file, of the first byte to be written. It must be a multiple of the sector size (128, 256, 512, or 1024).

psDataRet

is the memory address of the word to which the count of bytes successfully written is to be returned.

Request Block

ssDataRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	slotNumber*	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	12326
12	fh	2	
14	lfa	4	
18	baBuffer	4	
22	sBuffer	2	
24	psDataRet	4	
28	ssDataRet	2	2

*The slot number of the board from which the data is to be written

WriteRsRecord (pRswa, pRecord, sRecord): ercType

Description

WriteRsRecord writes a record to the open RSAM file identified by the memory address of the Record Sequential Work Area. The RSAM file is automatically extended to accommodate new records. Because output is buffered, there is no guarantee of the time at which output is actually written. Only the CheckpointRsFile and CloseRsFile operations ensure that data was actually written.

Procedural Interface

WriteRsRecord (pRswa, pRecord, sRecord): ercType

where

pRswa

is the memory address of the same Record Sequential Work Area that was supplied to OpenRsFile.

pRecord
sRecord

describe the memory area containing the record to be written.

Request Block

WriteRsRecord is an object module procedure.

This page intentionally left blank

WriteStatusC (pBs, wStatusMask, wStatus): ercType

Description

WriteStatusC writes to the specified communications lines status bits, changing the condition of the corresponding status lines. (See "Communications Programming" in the *CTOS Operating System Concepts Manual.*)

This operation blocks (waits) while there are characters still in the transmit buffer. If blocking must be avoided, call CheckpointBsAsyncC first.

wStatusMask is one of the following masks:

Mask	**Description**
0100h = DTR	(Data Terminal Ready)
0200h = RTS	(Request To Send)
0400h = STX	(Secondary Transmit)

Masks may be ORed together to reference more than one status bit at a time. Only the status bits selected by *wStatusMask* are accessed. The others are not written.

Procedural Interface

WriteStatusC (pBs, wStatusMask, wStatus): ercType

where

pBs

is the memory address of the Byte Stream Work Area (BSWA).

wStatusMask

is a word containing the selected mask bits, from the list above.

wStatus

is a word containing the new values of the status indicators, in the same bit positions as the corresponding mask bits.

Request Block

WriteStatusC is an object module procedure.

XbifVersion (pVerStruct, sVerStruct, pcbRet): ercType

Description

XbifVersion returns a structure that contains the version of the installed X-Bus Interface Service.

Procedural Interface

XbifVersion (pVerStruct, sVerStruct, pcbRet): ercType

where

pVerStruct
sVerStruct

describe a memory area to which the structure containing the version is written. The format of the structure is as follows:

Offset	**Field**	**Size (Bytes)**
0	version	2

pcbRet

describes a word into which the length of the returned structure is placed.

Request Block

scbRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	1	
6	exchResp	1	
8	ercRet	1	
10	rqCode	1	8217
12	reserved	6	
18	pVerStruct	4	
22	sVerStruct	2	
24	pcbRet	4	
28	scbRet	2	2

ZoomBox (pBoxDesc, sCols, sLines, fFinalBox): ercType

Caution: *Only one box at a time can be zoomed.*

Description

ZoomBox creates a new box or increases the size of an existing box in a video display. The operation expands the box from the screen center until the box dimensions specified by the values of *sCols* and *sLines* are reached. If *fFinalBox* is TRUE, a border is drawn around the box perimeter.

The box can be created using the description of an existing box that was created using the ZoomBox operation, or it can be created at the time this call is made. To create a box, ZoomBox obtains information from the box descriptor *pBoxDesc*. If the field *sBkgFrame* in the descriptor is not 0, calling the UnzoomBox operation restores the original screen after collapsing a box that ZoomBox originally created.

If the fields *iFrame*, *nCols*, and *nLines* define an existing box, ZoomBox expands out from the edges of the box to the specified dimensions of the new, larger box.

One application of ZoomBox is displaying customized user screens by a software installation program.

Enls operations should call QueryZoomBoxSize to determine the amount of memory to use for saving character and attribute data.

Procedural Interface

ZoomBox (pBoxDesc, sCols, sLines, fFinalBox): ercType

where

pBoxDesc

is the memory address of the box descriptor. If *nColsStart* and *nLinesStart* are 0, ZoomBox creates a new box starting at the screen center. The box descriptor has the following format:

Field	Size (Bytes)	Description
iFrame	2	is the frame number of an existing video frame, which is large enough to contain the box.
reserved	4	
nColsStart	2	is the width in columns of the initial box or is 0.
nLinesStart	2	is the height in lines of the initial box or is 0.
pBkgFrameChars	4	is the memory address of a buffer in which the characters of the existing character map are saved (so they can be restored by a call to UnzoomBox).
pBkgFrameAttrs	4	is the memory address of a buffer in which attributes from the existing character map are saved.
sBkgFrame	2	is the size in bytes of the character and attribute buffers described above. The size of each buffer is the same value. If *sBkgFrame* is 0, the character map of the original screen is not saved.

Field	Size (Bytes)	Description
pbTitle	4	is the memory address of a title string that appears at the top of the new box.
cbTitile	2	is the count of bytes in the title string.
fCenterTitle	1	is a flag that is TRUE if the box title is centered and FALSE if left justified.

sCols

is the width (word) in columns of the box to be created.

sLines

is the height (word) in lines of the box.

fFinalBox

is a flag (byte) that is set to TRUE if a border is to be drawn around the box. If *fFinalBox* is FALSE, the box is expanded to the specified dimensions, but the border is not drawn.

Request Block

ZoomBox is an object module procedure.

This page intentionally left blank

ZPrint (prgbString)

NOTE: This operation does not return a status code.

Description

ZPrint prints a null-terminated string to the video or other device as specified by the NPrint and PutChar operations.

Procedural Interface

ZPrint (prgbString)

where

prgbString

is the memory address of a character string terminated with a null (0) byte.

Request Block

ZPrint is an object module procedure.

This page intentionally left blank

43574565-100